

Boosting Quantum Classifier Efficiency through Data Re-Uploading and Dual Cost Functions

Supplementary Documentation

Authors: Sara Aminpour^{1,2}, Mike Banad¹, Sarah Sharif^{1,2*}

Author Affiliations:

¹ School of Electrical and Computer Engineering, University of Oklahoma, Norman, OK 73019, USA

² Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA

*Corresponding author: Sarah Sharif (email: s.sh@ou.edu)

Table of contents

- Supplementary Note 1. Range of training samples and number of layers
- Supplementary Note 2. Evaluating LCP and non-LCP approaches for fidelity cost function in fixed and random datasets for 1-qubit classifier for four different minimization methods
- Supplementary Note 3: Evaluating LCP and non-LCP approaches for trace distance cost function in fixed and random datasets for 1-qubit classifier
- Supplementary Note 4: performance comparison of 5-layer single-qubit quantum classifiers using fidelity and trace distance cost functions across various classification tasks and dataset types
- Supplementary Note 5: Evaluating LCP and non-LCP approaches for fidelity in fixed and random datasets for 2-qubit and 2-qubit entangled classifiers
- Supplementary Note 6: Method
- Supplementary Note 7: Optimization Methods
- Supplementary Note 8: Comparing our developed code with original reference

Supplementary Note 1: Range of training samples and number of layers

Figure S1.1 illustrates the performance of a quantum classifier utilizing a fidelity cost function within a five-layer framework for circular pattern classification in a fixed dataset, employing the L-BFGS-B optimization method. The analysis encompasses training data up to 250 samples to benchmark our algorithm against the findings from the reference¹. The diagram depicts training accuracy with a blue dashed line and test accuracy with a solid blue line, underscoring the algorithm's efficacy. A red dot highlights a notable benchmark from the reference, showing an 89% accuracy with 200 training samples, demonstrating parity with this published result. The inset provides a visual representation of the classification process. Notably, test accuracy begins at approximately 70%, rising impressively to 96% for a slightly expanded dataset of 210 samples. Remarkably, with as few as 60 training samples, the model achieves a test accuracy of 91.8%, and the discrepancy between training and test accuracy diminishes with the inclusion of 90 samples. This observation underscores the efficiency of our approach, highlighting its capability to reach high accuracy levels without necessitating extensive training data.

Figure S1.2 showcases a systematic evaluation of a circular pattern classification model across a spectrum of architectural depths, ranging from 1 to 5 layers. The graphical analysis reveals that models with a solitary layer lag in performance compared to those with increased layer counts, marking a clear trend: as the number of layers escalates, so does the model's classification accuracy. Specifically, a single-layer setup achieves a peak accuracy of 61.9%, whereas a more complex five-layer configuration significantly elevates this metric to 88.8%, even when limited to only 35 training samples. This observation underscores a critical insight—enhancing the model's depth systematically improves its predictive capabilities, a phenomenon consistent with the advantages afforded by the data reuploading strategy integral to our approach. Given this marked improvement in model efficacy with layer augmentation, the paper prioritizes an in-depth investigation and discourse on the five-layer model's architecture, focusing on its ability to optimize classification accuracy with efficient utilization of training data.

Supplementary Note 2: Evaluating non-linear and linear classification approaches for fidelity cost function in fixed and random datasets for 1-qubit classifier for four different minimization methods

Figure 2 illustrates a comparison of four distinct optimization techniques, namely L-BFGS-B, COBYLA, Nelder-Mead, and SLSQP, applied to the task of classifying the circle pattern. The comparison evaluates both training and test accuracies using a fixed dataset of 4000 test samples and 5 layers. Initially, all algorithms demonstrate a perfect training accuracy of 100% with just a single sample, a result that aligns with expectations. However, as we increase the sample size, a divergence in performance becomes evident for these four minimization methods. The L-BFGS-B method maintains a training accuracy close to 90%, showcasing its robustness against overfitting. In contrast, COBYLA, Nelder-Mead, and SLSQP show significant variability and a decline in training accuracy, indicating a susceptibility to overfitting. Interestingly, the peak accuracy for COBYLA, Nelder-Mead, and SLSQP is achieved with merely 50 samples, beyond which overfitting becomes a significant issue. This observation suggests that, unlike L-BFGS-B, which requires a minimum of 100 samples to achieve an accuracy of 92%, the other three methods can attain over 95% accuracy with only 50 samples. L-BFGS-B does not reach this high accuracy level at 100 samples, and its performance slightly declines with an increase in training samples after 150 training samples. This analysis highlights the critical importance of carefully selecting the number of training samples based on the minimization method used. The right choice can effectively prevent overfitting, thereby enhancing classification accuracy. This insight is crucial for optimizing machine learning models and ensuring their generalizability and efficiency in practical applications.

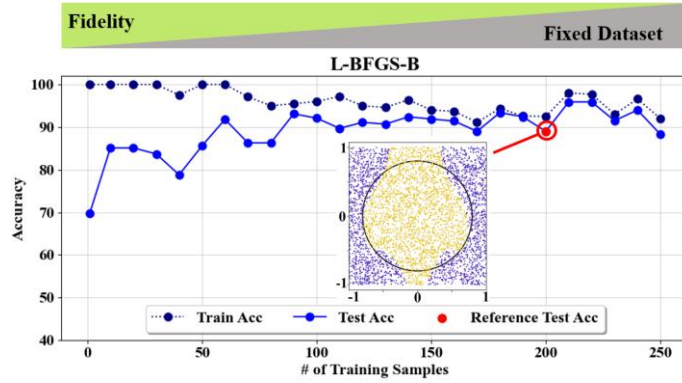


Figure S1.1 Train and test accuracy of fidelity for the 5-layer model of circle classification and fixed dataset for L-BFGS-B minimization method. The inset graph shows the visualization of a nonlinear classification reported on¹.

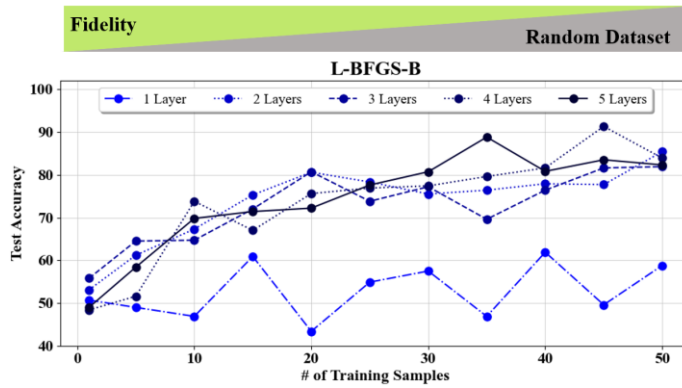


Figure S1.2. Evaluate the test accuracy of fidelity for circle classification and random dataset for L-BFGS-B minimization method, ranging from 1 to 5 layers.

Figure 3 delves into the accuracy of these four distinct minimization methods —L-BFGS-B, COBYLA, Nelder-Mead, and SLSQP— when applied to a fidelity cost function and a random dataset for circle classification. This analysis underscores a consistent trend across all methods: an initial increase in test accuracy corresponding to the rise in the number of training samples, yet fails to surpass a peak accuracy of 90%. This trend highlights the inherent challenges faced by these minimization methods when dealing with random datasets. In the L-BFGS-B method as depicted in figure 3(a), showcases a notable performance, achieving its highest test accuracy of 88.8% with 35 training samples. This point also marks the narrowest gap of 5% between training and test accuracy, indicating a relatively high level of model efficiency and generalization at this sample size. However, as the analysis progresses, it becomes apparent that increasing the number of training samples beyond this optimal point does not translate to improved performance. The gap between the train and test accuracy remains notably constant at around 10% even as the sample size is increased to 70 training samples. Transitioning to the COBYLA method, as depicted in figure 3(b), a different performance pattern emerges. Contrary to L-BFGS-B, COBYLA achieves its best test accuracy at 84.8% with a higher training sample equal to 70. This method experiences fluctuations, yet it is noteworthy that the gap between training and test accuracies exhibits a decreasing trend, suggesting a gradual improvement in model generalization compared to the initial stability seen with L-BFGS-B. Figure 3(c) focuses on the Nelder-Mead method, highlighting a decrease in the gap between training and test accuracies as the number of training samples increases, culminating in a maximum accuracy of 86.9% with 60 training samples. Figure 3(d) examines the SLSQP method, which shows an increase in test accuracy up to 50 training samples before demonstrating a decline in both training and test accuracies. This shows the SLSQP method is more prone to overfitting. The SLSQP method reaches a maximum accuracy of 86.7% when applied to a dataset of 50 samples. These results, as detailed in figure 6, provide vital insights into the performance of various minimization methods when working with a fidelity cost function and a random dataset. The diverse outcomes emphasize the importance of choosing an optimal number of training samples to prevent overfitting and enhance accuracy. This underlines the delicate balance needed to fully leverage these computational methods in practical scenarios.

Figure 4 illustrates a comparison of four different optimization techniques applied to the task of classifying line patterns, using fidelity-based cost function and the fixed dataset. The subplot (a) focuses on the performance of the L-BFGS-B method. Here, the training accuracy starts at a perfect 100% and impressively remains above 97% even as the number of training samples increases. Conversely, the test accuracy initiates at a relatively lower rate of 62.2% with just a single sample yet it progressively improves, reaching approximately 95% accuracy with 75 training samples and slightly declines for larger training samples. An initial notable gap between the training and test accuracy is evident, but this gap diminishes significantly as the dataset expands with more training data, indicating an improvement in the model's ability to generalize from the training to the unseen test data. The subplot (b) depicts the results obtained using the COBYLA algorithm, which exhibits a performance pattern similar to that of the L-BFGS-B method, consistently achieving 100% accuracy on the training data. The accuracy on the test set starts at 66.9% and steadily improves as more training samples are added, ultimately reaching 95% when 125 samples are used for training. The disparity between training and test set accuracies mirrors the pattern observed with the L-BFGS-B method, consistently manifesting across all training dataset sizes. The Nelder-Mead approach, shown in figure 4(c), achieves a notable test accuracy of 97.7% with 125 training samples. The inset provides a graphical visualization of line classification using this minimization method at this specific point, illustrating that the line classification performance is exceptionally well. The visualization clearly demonstrates the method's effectiveness in accurately separating the data points into distinct classes, highlighting the Nelder-Mead method's precision and robustness in handling line classification tasks with a substantial number of training samples. Furthermore, the training and test accuracy curves show a notably smaller gap, converging to the same value with training sets of 100 and 125 samples. The final subplot (d) evaluates the performance of the SLSQP method, which closely aligns with the results from the COBYLA method. The test set accuracy exhibits a progressive increase, rising from 62.7% to 96.6%. The disparity between the training and test accuracies is similar to that observed with the COBYLA method. In summary, all four optimization techniques demonstrate a reduction in overfitting as the training dataset size increases, ultimately achieving a test accuracy of at least 95% when training with 125 samples for this line classification task.

Figure 5 showcases an analysis of the classification accuracy obtained using the same minimization methods across random datasets. Consistently, a rise in the number of training samples correlates with an increase in test accuracy across all methods evaluated. Notably, with just 50 training samples, all methods surpass the 90% accuracy threshold. Specifically, in figure 5(a), the L-BFGS-B method reaches the peak accuracy of 92.8% with 50 training samples. It was observed that as the number of samples increased, the disparity between train and test accuracies for the L-BFGS-B method began to narrow, although this gap persisted in being slightly wider than that observed in the other methods. Figure 5(b) demonstrates that the COBYLA method, with the same number of samples, attains a superior accuracy of 93.5%. This suggests that COBYLA not only reaches high classification accuracy with a minimal dataset but also demonstrates better generalization compared to L-BFGS-B, as reflected by the narrower gap between its training and test accuracies. Figure 5(c) examines the Nelder-Mead method, showing its peak accuracy of 93% with 40 training samples, after which its accuracy slightly declines. Interestingly, the smallest disparity between training and test accuracies—only 1.8%—occurs in 50 training samples. Despite slightly lower accuracy at this point, this smallest gap signifies that the Nelder-Mead method achieves a remarkable balance between learning from the training data and generalizing to unseen data, highlighting its efficiency and potential for precise model tuning. Figure 5(d) illustrates that the SLSQP method achieves an impressive peak test accuracy of 96.4% for line classification using a random dataset, attained with 45 training samples. At this juncture, the discrepancy between training and test accuracies is notably small, indicating a high level of model precision and generalization. Like the Nelder-Mead method, the SLSQP method exhibits a nonmonotonic increment in test accuracy as a function of training samples, as indicated by the irregular slope of test accuracy. This fluctuation suggests that for these methods, adding more training samples does not straightforwardly translate to higher test accuracies, highlighting the complexity of optimizing model performance across different minimization techniques.

A comparison of figures 2 and 4 reveals that the accuracy curves for line classification are more stable and consistent across all optimization techniques when compared to those for circle classification. The accuracy values for classifying circle patterns display greater variability and fluctuations than those observed in the line classification task. The observed differences in performance between circle and line classification could stem from several technical factors: (1) Line classification likely represents a more straightforward pattern that aligns better with the linear decision boundaries most classifiers are adept at identifying. In contrast, circle classification involves recognizing more complex, non-LCP, which can challenge the classifiers' ability to generalize from the training data without overfitting or underfitting. (2) The algorithms applied for circle classification might be more prone to getting trapped in local minima due to the more intricate decision boundaries required to accurately classify circular patterns. This can hinder the optimization process, leading to increased fluctuations in classification accuracy as the model struggles to find the global optimum. (3) The differences in performance may also reflect the inherent adaptability of the algorithms to the specific types of classification tasks with the geometric properties. A comparative analysis of Figures 6 and 8 indicates that the specific characteristics of the classification problem significantly affect the potential to attain higher accuracy with fewer samples. The fluctuations in the line classification pattern are less pronounced than those in the circle classification pattern. This observation underscores the importance of selecting appropriate optimization methods based on the complexity of the classification problem.

Supplementary Note 3: Evaluating non-linear and linear classification approaches for trace distance cost function in fixed and random datasets for 1-qubit classifier

Figure 6 showcases the effectiveness of the trace distance cost function in classifying circular patterns within a fixed dataset. In subplot (a), the L-BFGS-B minimization method achieves its highest test accuracy at 79.2% with a dataset comprising 100 training samples. Subplot (b) examines the performance of the COBYLA method, which displays greater variability in training accuracy than L-BFGS-B but ultimately achieves a higher peak test accuracy of 84.6%, also with 100 training samples. Notably, COBYLA demonstrates enhanced generalization capabilities relative to other methods, as indicated by the narrower margin between its training and testing accuracies. This performance suggests that, when applied alongside the trace distance cost function, the COBYLA method is particularly adept at optimizing parameters for improved generalization to unseen testing data. An accompanying visualization within the inset illustrates the classification of circular patterns at this accuracy peak. In subplot (c), the analysis shifts to the performance of the Nelder-Mead method, which records its optimal test accuracy at 72.6% utilizing 60 training samples. This method exhibits signs

of overfitting, a condition where the model learns the training data too closely and fails to generalize well to new, unseen data. Despite a narrowing gap between training and testing accuracies as the number of training samples grows, a concurrent decline in training accuracy is observed, which adversely affects the overall test accuracy. This pattern suggests a limitation in the Nelder-Mead method's capacity to effectively handle the trace distance cost function, likely due to its inherent characteristics such as reliance on simplex-based optimization, which might struggle with the complexity of the trace distance landscape. Consequently, this method appears less suited for tasks requiring robust generalization from the trace distance cost function, particularly in scenarios demanding accurate classification of complex patterns with a limited dataset. In subplot (d), the focus turns to the SLSQP method which attains its peak test accuracy at 83.6% with a dataset of 100 training samples. The disparity between training and testing accuracy contracts by increasing the training samples, indicating an improvement in the model's ability to generalize from the training to the testing dataset. However, even at the point of 100 training samples, the gap between training and testing accuracies, while reduced, remains significant. This persistent gap suggests that while the SLSQP method is effective at learning and generalizing from the given data, there is still a margin for optimization to further bridge the difference in accuracies. Each optimization technique successfully minimizes the cost function and attains perfect accuracy on the training set using a comparatively small number of samples. However, their performance varies considerably when it comes to generalizing to the test set. This highlights the crucial role played by the choice of optimization algorithm in determining the overall effectiveness of the model. In conclusion, when considering the fixed dataset and the trace distance cost function, the COBYLA method demonstrates superior performance in optimizing the parameters to generalize effectively to unseen test data. Compared to the other techniques evaluated, it necessitates fewer training samples to achieve satisfactory accuracy on the test set.

Figure 7 illustrates how the accuracy on both the training and test sets evolves as the number of training samples grows, specifically for the task of classifying circular patterns using the trace distance cost function, evaluated on a randomly generated dataset. Similar to all scenarios analyzed so far, a common pattern emerges where test accuracy begins at a relatively low level for all minimization methods but demonstrates a consistent increase as more training data is provided. This trend highlights the methods' capacity to effectively learn distinguishing features, thereby enhancing their ability to generalize to unseen data. Specifically, in subplot (a), the L-BFGS-B method illustrates impressive learning efficiency, with test accuracy exceeding 70% after incorporating just 40 training samples and achieving its highest test accuracy of 77.8% with 45 training samples. In subplot (b), the COBYLA method's performance is slightly lower compared to L-BFGS-B, plateauing at a test accuracy of 72.8% with 45 training samples. This performance indicates that while COBYLA may be susceptible to some degree of overfitting, it nonetheless achieves a reasonable level of generalization. Subplot (c) explores the Nelder-Mead method, which reaches its peak test accuracy of 75.1% with 50 training samples. Subplot (d) utilizes the SLSQP method, which shows fluctuations in its training accuracy remaining above 80%. The test accuracy for SLSQP was enhanced significantly, reaching 74.6% with 50 samples. This fluctuation and eventual rise in test accuracy underscores the method's potential for optimizing classification tasks, despite the initial variability. In sum, the L-BFGS-B method stands out for achieving the highest test accuracy among the methods evaluated, requiring only 45 training samples to reach this optimum on a random dataset. Summarily, employing the trace distance cost function across these various minimization strategies yields test accuracy ranging from 65% to 78% on the random dataset, illustrating the function's effectiveness and the distinct performance capabilities of each minimization method.

Figure 8 offers a comparative analysis of the accuracy achieved by four different optimization methods when applied to a trace distance cost function for line pattern classification using a fixed dataset. Subplot (a) highlights the L-BFGS-B method, showcasing its high level of stability in training accuracy. The test accuracy shows a steady increase, reaching 91.8% with 100 training samples. While there is a substantial gap between the accuracies of the training and test sets at the outset, this difference gradually narrows as more training samples are introduced. This highlights the L-BFGS-B method's capacity to adapt and learn more complex patterns effectively, demonstrating robustness and in leveraging larger datasets for improved generalization. The subplot (b) illustrates the results obtained using the COBYLA method. In contrast to the L-BFGS-B approach, the accuracy of the training set shows greater fluctuations, even experiencing a drop to 56.9% at one instance before rebounding. The test accuracy follows a similar pattern to that seen in L-BFGS-B,

beginning at 49.8% and increasing to 87.4%. Once the training set size reaches 80 samples, both the training and test accuracies seem to reach a plateau, slightly below the 90% mark. In subplot (c), the Nelder-Mead method starts with a modest test accuracy of 55.3%, which significantly improves to 87% with the addition of 60 training samples demonstrating a similar trend as the L-BFGS-B method. Initially, a pronounced gap exists between training and test accuracies, which persists until the dataset is expanded to include 80 training samples. Beyond this point, the sign of overfitting emerges, as demonstrated by a decline in training accuracy while test accuracy plateaus. For 100 training samples, the test accuracy interestingly becomes 2% higher than the training accuracy, indicating a unique inversion where the model performs slightly better on unseen data than on the training set itself, a rare occurrence that may suggest the model has reached a point of optimization where it generalizes exceptionally well to new data. The subplot (d) of figure 11 presents the results of the SLSQP method. Notably, this technique achieves the highest accuracy on the test set, reaching 93.3% using just 40 training examples. The SLSQP method appears to be the most appropriate choice for trace distance classification tasks, as it exhibits a smaller discrepancy between its performance on the training and test datasets. The inset provides a visual representation of the SLSQP's performance at this specific point. To summarize, all optimization methods demonstrate an upward trajectory in test accuracy as the size of the training dataset increases, suggesting enhanced generalization capabilities of the model. Among the four techniques evaluated, the SLSQP method seems to strike the most favorable balance between its performance on the training and test sets.

Figure 9 presents a comparison of different optimization techniques when applied to the task of classifying line pattern using a randomly generated dataset and a cost function based on trace distance. In subplot (a), we examine the performance of the L-BFGS-B method, which attains its peak test accuracy of 86.3% with 55 training samples. Before reaching this point, the method's test accuracy demonstrated considerable variability, oscillating between 70% and 80% as the number of training samples ranged from 20 to 50. However, a notable improvement occurs when the dataset is expanded to 55 training samples, at which the test accuracy leaps to 86.3%, effectively surpassing the earlier fluctuation band. This pivotal moment also marks the occurrence of the smallest gap between training and test accuracies, showcasing a significant enhancement in the model's ability to generalize from the training dataset to unseen data, thereby achieving an optimal balance at this specific training sample size. Subplot (b) delves into the efficacy of the COBYLA optimization method, which achieves its highest test accuracy of 86.8% with a relatively smaller dataset of 35 training samples. Beyond this optimal threshold, signs of overfitting become apparent, as both training and test accuracies start to decline. This pattern suggests that while the COBYLA method is highly effective up to a certain point, adding more training samples beyond this number paradoxically hampers the model's performance. The decline in accuracy indicates that the model begins to memorize the training data rather than learning to generalize, leading to a decrease in its ability to accurately predict outcomes on unseen data. This observation underscores the importance of identifying the ideal number of training samples to maximize the effectiveness of the COBYLA method without crossing into the territory of overfitting. In subplot (c), the focus is on the Nelder-Mead optimization method, which shows some fluctuations in performance before reaching its maximum test accuracy. It successfully achieves a test accuracy of 88.1% with 40 training samples. However, akin to the pattern observed with the COBYLA method, the Nelder-Mead method also sees a decline in both training and test accuracies when additional training samples are added beyond this optimal number. This decline serves as a clear indication of the onset of overfitting, suggesting that while the Nelder-Mead method can efficiently utilize a certain number of training samples to improve its predictive accuracy, exceeding this number leads to a reduction in model performance. In subplot (d), a more continuous and stable increase in test accuracy is observed with each increase in the number of training samples. This trend results in the highest test accuracy being recorded at 88.3% with 55 training samples. Unlike the previous methods discussed, this subplot suggests a method that maintains its efficiency and ability to generalize well without showing immediate signs of overfitting up to this point. The gradual and consistent improvement in test accuracy highlights the method's effective learning curve and suggests an optimal balance between learning from the training data and applying this knowledge to unseen data.

Supplementary Note 4: performance comparison of 5-Layer single-qubit quantum classifiers using fidelity and trace distance cost functions across various classification tasks and dataset types

Figure S4.1 offers a comparative analysis of the highest accuracies achieved for two distinct classification patterns – linear (line) and non-linear (circle) – across the four distinct minimization methods when applied to both random and fixed datasets within the context of a fidelity cost function. The analysis reveals a notable trend: in circle classification tasks, the fixed dataset consistently yields higher accuracies than their random counterparts for all tested minimization methods. This suggests that the inherent geometric complexities of non-LCP may align more closely with the simpler structure of fixed datasets, thereby facilitating more accurate classification. Similarly, for line classification, the fixed dataset leads to enhanced accuracies with the L-BFGS-B and SLSQP methods, indicating these methods' effectiveness in leveraging structured data to accurately discern linear relationships. However, the random dataset achieves better accuracy when classified using the Nelder-Mead method. This could suggest that the Nelder-Mead method, known for its simplicity and direct search approach, might be particularly adept at navigating the stochastic nature of random datasets to identify linear patterns. Across all algorithms, the task of classifying non-LCP, especially within random datasets, emerges as inherently challenging. This complexity likely stems from the algorithms' varying abilities to parse and learn from the unpredictable variance found in random datasets, as well as the added difficulty of accurately modeling non-linear relationships. The findings underscore the critical importance of selecting the appropriate minimization method based on the dataset's nature and the classification task's geometric complexity to optimize classification accuracy.

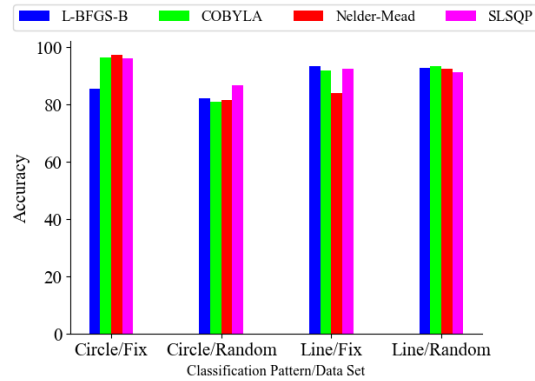


Figure S4.1. Evaluating of Fidelity cost function test accuracy of 5-layer model across 50 samples for LCP and non-LCP problems for random and fixed datasets in four minimization methods.

Figure S4.2 provides the performance comparison of two distinct classification patterns—line and circle—across four different minimization methods when applied to both random and fixed datasets, this time employing the trace distance cost function. A pivotal observation emerges when comparing the performance of circle classification with a fixed dataset (circle/fixed) against the fidelity cost function results presented in figure S4.1. It is evident that the accuracies achieved using the trace distance cost function are notably lower across all minimization methods compared to those obtained with the fidelity cost function. This discrepancy highlights the inherent challenges and differences in how each cost function interacts with the underlying data and the classification task at hand. The trace distance cost function, known for quantifying the distinguishability between quantum states, may present a more complex landscape for optimization, particularly when applied to classical data patterns such as lines and circles. This complexity could lead to lower classification accuracy as the minimization methods struggle to navigate the nuances of the trace distance landscape effectively. Such an observation underscores the importance of cost function selection in machine learning tasks, emphasizing that the

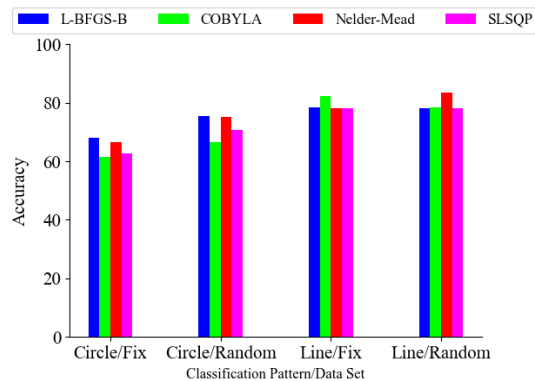


Figure S4.2. Evaluating of trace distance test accuracy of 5-layer model across 50 samples for LCP and non-LCP problems for random and fixed datasets in four minimization methods.

choice of cost function can significantly impact the model's ability to learn and generalize from the data. The comparative analysis in figure S4.2 serves as a testament to the nuanced interplay between cost functions, dataset types (fixed vs. random), and the geometric nature of the classification patterns, offering valuable insights into optimizing classification accuracy through strategic method and cost function selection.

In addition, the fixed dataset achieves superior accuracy specifically when employing the COBYLA minimization method, indicating a unique synergy between COBYLA's optimization strategy and the structured nature of fixed datasets for LCP. Conversely, for the random dataset, there's a notable trend where it consistently outperforms the fixed dataset across all other minimization methods, suggesting that the stochastic characteristics of random datasets may be better suited to the optimization landscapes these methods navigate, particularly for LCP. In circle classification tasks, the random dataset not only demonstrates improved accuracy over the fixed dataset for all minimization methods but also reinforces the observation that random datasets generally offer a more favorable context for the trace distance cost function across both classification patterns. This enhancement in accuracy with random datasets could be attributed to the trace distance cost function's sensitivity to the variances within the dataset, allowing for more effective differentiation and classification of non-LCP like circles when the data is less predictable.

Supplementary Note 5: Evaluating non-linear and linear classification approaches for fidelity in fixed and random datasets for 2-qubit and 2-qubit entangled classifiers

Focusing on figure 10(a), we observe the performance of a single-qubit system applied to a LCP pattern. The system demonstrates a steep initial learning curve, with accuracy rapidly increasing from 51.6% to 92% after just 75 training samples. This sharp rise highlights the single-qubit system's ability to efficiently learn and generalize from a relatively small dataset. The notable jump in accuracy suggests that a properly trained single-qubit classifier can capture the essential features of the LCP task with high precision. After reaching 92% accuracy at 75 training samples, the system stabilizes, maintaining a test accuracy consistently in the range of 92% to 97.7% as the training sample size increases to 125. The minimal fluctuation in accuracy indicates a robust performance, with the single-qubit system effectively avoiding overfitting even as the training data expands. The stable test accuracy underscores the system's reliability and suitability for LCP tasks where computational simplicity and consistent performance are crucial. In terms of computational cost, as shown in figure 1(d), the single-qubit system exhibits a gentle increase in computational time, reaching 62.15 seconds for 250 training samples. This computational efficiency, coupled with the system's stable accuracy, makes the single-qubit classifier an appealing option for linear problems, particularly in scenarios where computational resources are limited but high accuracy is still required.

In figure 10(b), the performance of the 2-qubit classifier in a LCP task shows a more gradual improvement in accuracy compared to the single-qubit system. The initial accuracy is relatively high, starting at 73.2% with just one training sample, which suggests that the additional qubit provides a more robust representation of the problem space even with minimal training. As the number of training samples increases to 75, the accuracy rises steadily, reaching 94.1%. This gradual improvement, as opposed to the sharp jump seen in the single-qubit system, highlights the ability of the 2-qubit classifier to build on its already strong initial performance with increasing training data. Beyond 50 training samples, the 2-qubit classifier continues to demonstrate incremental gains, eventually peaking at around 95.7% test accuracy with 175 training samples. Notably, the test accuracy fluctuates between 92% and 96% throughout this range, suggesting that while the system performs consistently well, there are slight variations in how the test data is classified as more training samples are introduced. These fluctuations could indicate that the system is sensitive to the nature of the training data or potentially approaching the limits of its capacity for linear classification. From a computational perspective, shown in Figure 1(e), the 2-qubit classifier exhibits a significant increase in computational time as the number of training samples grows. By the time the training sample size reaches 250, the computational time extends to around 260 seconds. This is a sharp contrast to the single-qubit system, illustrating the tradeoff between the enhanced accuracy and robustness offered by the 2-qubit classifier and the increased computational demands. For LCP tasks, this suggests that while the 2-qubit classifier provides higher initial accuracy and steady performance improvements, it comes at the cost of a much higher computational burden, making it potentially less suitable for scenarios where time or resources are constrained.

Examining figure 10(c), the performance of the 2-qubit entangled classifier in a LCP task reveals a distinctive pattern when compared to non-entangled systems. The initial accuracy is relatively low, starting at 51.3% with just one training sample. This suggests that the entanglement introduces complexities that make the system less effective in identifying patterns from very limited data. However, as the number of training samples increases to 75, the system exhibits a steep improvement in accuracy, reaching 93.3%. This rapid climb indicates that while the entangled system may struggle with very small datasets, it quickly capitalizes on additional training samples to enhance its classification performance. As the training samples continue to increase beyond 75, the 2-qubit entangled classifier shows notable fluctuations in accuracy, ranging between 88% and 97.5%. These fluctuations, which are more pronounced than those seen in the single-qubit or non-entangled 2-qubit classifier, suggest that entanglement introduces both benefits and challenges. On one hand, the system achieves the highest peak accuracy (97.5%) among all three systems, demonstrating its potential for superior performance. On the other hand, the variability in test accuracy highlights the sensitivity of the entangled system to the training data, possibly indicating overfitting or instability when processing larger datasets. In terms of computational cost, as shown in figure 10(f), the 2-qubit entangled classifier mirrors the trend seen in the non-entangled 2-qubit classifier, with computational time increasing significantly as the number of training samples rises. At 250 training samples, the computational time reaches 260 seconds, similar to the non-entangled classifier. Despite this computational burden, the 2-qubit entangled classifier offers a potential advantage in terms of peak accuracy, making it a compelling choice for applications where achieving the highest possible accuracy is paramount, even if it comes with the tradeoff of greater computational complexity and variability in performance.

In comparing the classifier, we observe clear tradeoffs between simplicity, stability, and computational complexity. The single-qubit classifier is the most stable and computationally efficient but may not reach the same peak accuracies as the more complex systems. The 2-qubit classifier offers higher initial accuracy and consistent improvement but requires significantly more computational resources. Finally, the 2-qubit entangled system, while achieving the highest peak accuracy, also introduces greater instability and computational demands, making it best suited for scenarios where peak performance is the priority, and computational cost is less of a concern. Ultimately, the choice of system depends on the specific requirements of the classification task, such as whether stability, computational efficiency, or peak accuracy is the primary objective.

Figure 11 presents a comprehensive analysis of two quantum classifiers - a 2-qubit classifier and a 2-qubit entangled classifier for non-LCP. The results are displayed across six subplots, labeled (a) through (f), which provide insights into the performance and characteristics of these classifiers under various conditions. Subplots (a) and (b) show the train and test accuracies as a function of the number of training samples for the 2-qubit and the 2-qubit entangled classifiers, respectively. In both cases, we observe that the accuracies generally improve as the number of training samples increases. However, the 2-qubit classifier (a) shows higher initial test accuracy, 73.5%, and a more stable performance across different sample sizes. The 2-qubit entangled classifier (b) starts with lower test accuracy, 47.6% but shows significant improvement as the sample size increases. Both classifiers seem to converge in terms of train and test accuracy around 175 training samples, which explains why this number was chosen for subsequent analyses. Subplots (c) and (d) illustrate how the number of layers in the quantum circuit affects the accuracies of the classifiers for a specific number of training samples. For the 2-qubit classifier (c), we see a general upward trend in both train and test accuracies as the number of layers increases, with some fluctuations. The 2-qubit entangled classifier (d) shows a more pronounced improvement with increasing layers, especially in the early stages. Both classifiers appear to reach a plateau in performance after about 12-15 layers, suggesting that further increases in circuit depth may not yield significant improvements. Subplots (e) and (f) depict the computational time required as the number of layers increases for the 2-qubit and the 2-qubit entangled classifiers, respectively. Both show a clear exponential growth in computational time as the number of layers increases. This trend is consistent across both classifiers, indicating that the computational cost scales similarly regardless of whether entanglement is used. Comparing the classifiers overall, we can see that the 2-qubit classifier generally achieves higher accuracies with fewer training samples and maintains more consistent performance across different numbers of layers. The 2-qubit entangled classifier, while starting with lower accuracy, shows more dramatic improvements as both the number of training samples and layers increase. This

suggests that entanglement might provide additional expressive power to the classifier, allowing it to capture more complex patterns in the data as the circuit depth increases. However, this potential advantage comes at the cost of increased sensitivity to the number of training samples and layers, as evidenced by the more volatile accuracy curves in subplots (b) and (d). The computational time plots (e) and (f) remind us that increasing the number of layers quickly becomes computationally expensive for both classifiers, which is an important consideration in practical applications. In conclusion, these results provide valuable insights into the trade-offs between accuracy, circuit complexity, and computational cost for quantum classifiers, highlighting the potential benefits and challenges of using entanglement in quantum machine learning tasks.

Figure 12 presents a comparative analysis of four optimization algorithms (COBYLA, L-BFGS-B, NELDER MEAD, and SLSQP) applied to a LCP using a quantum circuit with 2 qubits. The experiment uses a random dataset with 250 training samples and employs a fidelity cost function to measure the performance. The figure includes subplots depicting accuracy and computational time for both 2-qubit and 2-qubit entangled classifiers. In terms of accuracy, both training and test accuracies are generally high across all algorithms. However, there are subtle differences between the algorithms. As shown in figure 12(a), for the 2-qubit entangled classifier, the average test accuracy is approximately 2% higher than the 2-qubit non-entangled classifier. In terms of individual performance, the L-BFGS-B minimization method consistently achieves the highest test accuracy, reaching 96.3% for non-entangled and 97% for entangled classifiers. The overall variation in test accuracy between the highest and lowest performing algorithms is 2.3%. For 2-qubit non-entangled classifier, COBYLA exhibits the lowest test accuracy at 94%, while for 2-qubit entangled classifier, NELDER MEAD achieves the lowest test accuracy of 95.3%. Computational time analysis reveals interesting patterns across both classifiers. In figure 12(c) the 2-qubit classifier, computational time varies widely from 9 to 90 minutes. COBYLA stands out as the fastest method, completing the task in just 9 minutes, while L-BFGS-B and NELDER_MEAD are the most time-consuming at 90 and 89 minutes respectively. SLSQP occupies a middle ground, requiring 45 minutes. In figure 12(d) the 2-qubit entangled classifier generally shows improved computational efficiency. While COBYLA maintains its swift performance at 9 minutes, other methods see reduced execution times. Most notably, L-BFGS-B improves from 90 to 71 minutes, a significant reduction, while NELDER_MEAD and SLSQP methods remain at 87 and 44 minutes respectively. In conclusion, this analysis reveals that the 2-qubit entangled classifier generally outperforms the 2-qubit non-entangled classifier in both accuracy and computational efficiency. The L-BFGS-B method consistently provides the highest accuracy, albeit at a higher computational cost. COBYLA emerges as a well-balanced option, offering good accuracy with minimal computational time, particularly in the 2-qubit entangled classifier. These findings underscore the significant impact of minimization method selection on both accuracy and computational time in quantum machine learning tasks. Furthermore, the 2-qubit entangled classifier's closer alignment of train and test accuracies suggests enhanced generalization capabilities, a crucial factor in practical machine learning applications.

Figure 13 shows a comprehensive comparison of different optimization methods for non-LCP using both 2-qubit and 2-qubit entangled classifiers for a specific random dataset. This analysis encompasses four optimization techniques: COBYLA, L-BFGS-B, NELDER_MEAD, and SLSQP, evaluating their performance based on accuracy and computational time for 250 number of training samples. In the accuracy graphs (a) and (b), we observe distinct performance patterns between the 2-qubit and 2-qubit entangled classifiers. For the 2-qubit classifier, L-BFGS-B demonstrates the highest accuracy, with both train and test accuracies exceeding 90%. COBYLA shows the lowest performance, with a test accuracy of 76.7% and train accuracy 81.4%. NELDER_MEAD and SLSQP exhibit intermediate performance, with test accuracies in the 82-87% range. The 2-qubit entangled classifier, depicted in graph (b), shows overall improved accuracy across all methods. L-BFGS-B maintains its superior performance, while COBYLA shows significant improvement, reaching accuracies to 85.4%. Notably, the gap between train and test accuracies is generally smaller in the 2-qubit entangled classifier, suggesting better generalization. The computational time graphs (c) and (d) reveal interesting efficiency patterns. In the 2-qubit classifier, COBYLA is the fastest method, requiring only 9 minutes. L-BFGS-B, despite its high accuracy, is the most time-consuming at 130 minutes. NELDER_MEAD takes 89 minutes, while SLSQP requires 45 minutes. The 2-qubit entangled classifier (graph d)

shows generally reduced computational times. COBYLA remains the fastest, maintaining its 9-minute runtime. L-BFGS-B shows the most dramatic improvement, reducing its time to 81 minutes. Interestingly, NELDER_MEAD in the 2-qubit entangled classifier takes slightly longer than L-BFGS-B, at 88 minutes. SLSQP maintains a consistent performance of about 42 minutes in both systems. These results highlight the trade-offs between accuracy and computational efficiency in quantum machine learning tasks. The 2-qubit entangled classifier demonstrates superior performance in both accuracy and computational time across all methods. L-BFGS-B consistently provides the highest accuracy but at a higher computational cost, especially in the 2-qubit classifier. COBYLA emerges as a balanced option, offering good accuracy with minimal computational time, particularly in the entangled system. This analysis underscores the importance of choosing appropriate optimization methods and leveraging entanglement to enhance the performance of quantum classification tasks.

Supplementary Note 6: Method

Quantum computing manipulates quantum systems to enhance information processing, leveraging superposition to simultaneously operate on multiple states for faster and more complex computation. At its core is the qubit, represented in a two-dimensional Hilbert space, with operations governed by quantum gates. These gates, essential for altering quantum states, must be unitary to ensure the conservation of probability, a fundamental principle of quantum dynamics².

The framework of a quantum circuit unfolds in three key phases: encoding classical data into quantum format, manipulating the quantum state using quantum gates, and measuring the quantum state post-transformation. This process transitions from preparing an initial quantum state, through strategic alterations via quantum gates affecting computation outcomes, to a final probabilistic measurement—distinguishing quantum computing's potential and challenges from deterministic classical computing.

Achieving optimal performance in quantum computing requires a nuanced understanding of these phases, including the initial state preparation, the strategic selection and application of quantum gates, and the final measurement process. Each component must be meticulously optimized to perform specific tasks, such as classification, highlighting the intricate interplay between quantum mechanics and computational logic in the design and execution of quantum algorithms.

A. RE-UPLOADING CLASSICAL INFORMATION AND PROCESSING

The integration of classical information into quantum computing represents a groundbreaking approach to data processing and analysis. This process begins with the strategic encoding of data into the initial wave function's coefficients within a quantum circuit³. In simpler terms, data is initially uploaded through the manipulation of qubits via rotational operations on a computational basis. This foundational step sets the stage for executing sophisticated quantum algorithms, including those designed for classification tasks.

The most successful programming paradigm in machine learning is predicated on artificial neural networks, which represent a highly abstracted and simplified model inspired by the human brain⁴. An artificial neural network comprises interconnected units or nodes known as artificial neurons, often arranged in layers⁵. These networks are characterized by their diverse architectures and the ability to learn from data through the adjustment of a vast network of parameters during the training phase. Among the various types of neural networks, feed-forward neural networks exemplify the process of sequential data processing, where input data is transformed layer by layer, simulating a form of data re-uploading at each neuron. This mechanism of data re-uploading and processing in ANNs provides a parallel to the innovative approach of constructing a universal quantum classifier using a single qubit. The essence of this quantum computing strategy lies in the repeated introduction of classical data at different stages of computation, analogous to the data processing in a single hidden layer neural network. This process can be visualized diagrammatically, as shown in figure 14 in the main paper. the neural network architecture is depicted, where data points are fed into individual processing units, analogous to neurons within the hidden layer. These neurons collectively process these input data, culminating in the activation of a final neuron responsible for constructing the output for subsequent analysis. Similarly, in the quantum domain, the single-qubit classifier incorporates data points into each stage of the computation through unitary rotations. These rotations are not isolated; rather, each one builds upon the transformations applied by its predecessors, effectively integrating the input data multiple times throughout the computation. The culmination of this process is a quantum state that encapsulates the computational outcome.

To construct a universal quantum classifier with only a single qubit, a complex integration of data input and computational processing within a single quantum circuit is crucial. We achieve this objective through the deployment of parametrized quantum circuits (PQCs). In these circuits, certain rotational angles are meticulously adjusted based on

classical parameters, which are refined through an optimization process aimed at minimizing a specifically defined cost function.

The cost function plays a pivotal role in the operational efficacy of the quantum classifier. It quantitatively assesses the circuit's performance in segregating data points into distinct categories, which are represented as separate regions on the Bloch sphere. Each of these regions corresponds to a different class, and the classifier's goal is to assign data points to the correct class based on their features.

B. Dataset Generation Methodology

In this section, we provide a detailed and standardized description of how both fixed and random datasets were constructed and evaluated throughout the study.

- **Sampling Distribution and Dimensionality:**

All data points were sampled independently and uniformly from the interval $[-1,1]^2$, corresponding to the two-dimensional input space used in all classification problems. The sampling was performed using `np.random.rand(2)` and scaled via the transformation $x \mapsto 2x - 1$ to ensure full coverage of the $[-1,1]$ range along both axes.

- **Class Balance and Geometric Design:**

We carefully selected geometric parameters to maintain balanced class distributions. In the circle classification task (non-LCP), we used a radius of $r = \sqrt{2/\pi}$ such that the area inside and outside the circle is equal, yielding a 50/50 class distribution. For the linear classification task (LCP), we defined the decision boundary as $x_1 = x_2$, which symmetrically divides the domain $[-1,1]^2$ and likewise ensures class balance by design.

- **Reproducibility and Standardization:**

To ensure consistency across experiments, we fixed the random seed at 30 for all fixed dataset runs. The training set sizes varied from 1 to 200 samples depending on model complexity, while each test set consisted of 4000 uniformly sampled points. For randomized datasets, we deliberately omitted the use of a fixed seed, ensuring that each of the 20 iterations generated a new sample set from the same distribution. This approach allowed us to test the classifier's generalization ability and robustness under different data realizations. Accuracy and runtime were averaged across these 20 independent runs to obtain statistically meaningful results.

- **Dataset Types and Parameters:**

We focused on two primary classification tasks: (1) a line, representing linear separability (LCP), and (2) a circle, representing a basic non-linear separability case (non-LCP). These were chosen as fundamental and interpretable decision boundaries to evaluate the baseline performance of the quantum classifiers. All geometric parameters, such as the radius for non-LCP and the slope/intercept for LCP, were held fixed across all trials to ensure consistency and enable fair comparison across circuit designs and optimization methods.

C. Applying Cost Functions

In the realm of quantum computing, a quantum circuit is distinguished by its processing angles θ_i and associated weights w_i , leading to the generation of a final state $|\psi\rangle$. The measurement outcomes from this state are used to compute a classification error metric, defined as χ^2 . The goal is to minimize this error metric by adjusting the circuit's classical parameters, a process that can be effectively managed through various supervised machine learning techniques.

At the heart of using quantum measurement for classification tasks lies the approach of optimally aligning observed outputs with specific target classes. This alignment is primarily facilitated by the principle of maximizing orthogonality between the output states, ensuring clear distinction⁶. In the context of binary (dichotomous) classification, this means categorizing each observation into one of two predefined classes—referred to here as class A and class B. The decision criterion involves comparing the probabilities of observing the quantum state $P(0)$ for outcome 0 and $P(1)$ for outcome 1. If $P(0) > P(1)$, the observation is assigned to class A; otherwise, it falls under class B. To enhance this binary classification scheme, one can introduce a bias (λ), adjusting the threshold for classification such that observation is deemed part of class A if $P(0)$ is greater than λ , and class B if it falls below. The value of λ is chosen to maximize classification accuracy on a training dataset. The effectiveness of this approach is then confirmed through evaluation on a separate validation dataset.

Viewed through a geometric lens, the single-qubit classifier operates within a 2-dimensional Hilbert space—the Bloch sphere—where data encoding and classification decisions are delineated through specific rotational parameters. Any operation $L(i)$ is a rotation on the Bloch sphere surface. From this viewpoint, any point can be classified using just one unitary operation. Consequently, we can transfer any point to another point on the Bloch sphere by precisely selecting the rotation angles. However, when dealing with multiple data points, a single rotation may not suffice due to differing optimal rotation requirements. The solution lies in introducing additional layers into the quantum circuit, enabling distinct

rotation and fostering a richer feature map. Within this enhanced feature space, data points can be effectively segregated into their respective classes based on their positioning within the Bloch sphere's regions, thereby enabling a sophisticated and adaptable approach to quantum classification.

1) FIDELITY COST FUNCTION

The goal is to align the quantum states $|\psi(\vec{\theta}, \vec{w}, \vec{x})\rangle$ as closely as possible to a designated target state on the Bloch sphere, as outlined in ¹. This alignment can be quantitatively assessed by measuring the angular distance between the labeled state and the data state, using the metric of relative fidelity ⁷. The primary objective focuses on maximizing the average fidelity between the quantum states produced by the circuit and the label states corresponding to their respective classes. To facilitate this, a cost function is introduced and mathematically formulated as Equation 1:

$$\chi_f^2(\vec{\theta}, \vec{w}) = \sum_{\mu=1}^M \left(1 - \left| \langle \tilde{\psi}_s | \psi(\vec{\theta}, \vec{w}, \vec{x}_\mu) \rangle \right|^2 \right) \quad (1)$$

where $|\tilde{\psi}_s\rangle$ is the correct label state of the μ data point, which will correspond to one of the classes.

2) TRACE DISTANCE COST FUNCTION

In quantum information theory, quantifying the dissimilarity between two quantum states is a fundamental problem. Various distance measures have been proposed, each with its unique properties and applications. One such measure is the trace distance, which captures the distinguishability between two quantum states ⁷. Perez-Salinas et al. have analyzed the fidelity cost function with data re-uploading ¹. However, the authors do not consider the case of the trace distance cost function, which is what we focus on in this section. We will explore the definition and properties of the trace distance, particularly in the context of single-qubit systems, and discuss its potential as a cost function for quantum classifiers. Despite the different mathematical formulations of trace distance and fidelity, these two measures share many similar properties and are widely used in the quantum computing and quantum information community. However, depending on the specific application, one measure may be more convenient or easier to work with than the other. This versatility and widespread adoption of both trace distance and fidelity in the field motivates our decision to discuss and compare these two important distance measures in the context of quantum classifiers. The trace distance between quantum states ρ and σ can be defined as,

$$D(\rho, \sigma) \equiv \frac{1}{2} \text{tr} |\rho - \sigma|^2 \quad (2)$$

The trace distance between two single-qubit states, represented by their respective Bloch vectors \vec{r} and \vec{s} , is equal to one-half of the Euclidean distance between these vectors on the Bloch sphere. ⁷

$$D(\rho, \sigma) = \frac{|\vec{r} - \vec{s}|}{2}. \quad (3)$$

This relation provides a geometric interpretation of the trace distance for single-qubit systems, linking it to the intuitive notion of distance in three-dimensional space.

D. From Universality of the Single-Qubit Classifier to the Expansion into Multi-Qubit Quantum Classification

A key challenge in Quantum Machine Learning (QML) involves creating quantum circuits that efficiently handle complex tasks like classification without excessive use of quantum resources. The Universal Approximation Theorem (UAT) ⁸ is crucial for tackling this issue, demonstrating that a single-layer neural network with an appropriate activation function can approximate any continuous function to a desired accuracy, assuming enough hidden neurons are available. This UAT finds a compelling parallel in the quantum computing domain, particularly when considering the dynamics of quantum circuits. Here, the classical activation function is analogously performed by a unitary rotation acting upon a qubit. Specifically, a single-qubit quantum classifier, enhanced by the technique of data re-uploading, emerges as a universal approximator for any conceivable classification function. This universality hinges on the frequency of data re-uploading throughout the circuit's span ¹, underscoring that even a solitary qubit is capable of encoding and processing multifaceted high-dimensional data. This is achieved through the execution of multiple rotations, each characterized by distinct angles and weights. The culmination of these processes is a final quantum state, which is then analyzed against a predefined target state correlating to each class. Optimization of the circuit's parameters is pursued through the minimization of a cost function, which is indicative of the fidelity or trace distance between the comparative states.

By establishing the UAT within the context of quantum classifiers, a robust theoretical foundation is laid, alongside practical guidelines for the design and implementation of quantum circuits adept at sophisticated and non-LCP tasks with minimal quantum resource expenditure. This breakthrough not only forges a theoretical link between quantum circuits and neural networks but also paves the way for innovative approaches in QML. Through this lens, quantum circuits are envisioned not merely as computational tools but as entities with the potential to parallel, and possibly surpass, the capabilities of their classical neural network counterparts, inspiring a new wave of methodologies in the realm of QML.

To enhance the performance of the single-qubit classifier, it is proposed to extend it to a multi-qubit system. Adding more qubits, especially with entanglement, can improve the classifier's effectiveness, similar to how adding layers enhances neural networks. Entanglement may provide a quantum advantage in classification, though the analogy between multi-qubit classifiers and neural networks with entanglement is not fully understood and requires further exploration. Perez et al. propose a measurement strategy for multi-qubit classifiers, which extends the single-qubit approach. These strategies utilize a fidelity-based cost function.

E. Variational Circuit Architecture and Parameterization

To fully specify the architecture of the quantum classifier and support reproducibility, we detail here the structure of the variational circuits used in this study. The models are built using a data re-uploading framework, in which classical input data is embedded into the quantum circuit by modifying gate parameters via a linear transformation. Each circuit is composed of multiple layers; each layer includes data-dependent single-qubit gates followed by optional entanglement gates between qubits.

The primary quantum gates used are $U(\phi)$ gates, which are universal single-qubit rotation gates parameterized by three angles $\phi = (\theta, \varphi, \lambda)$. These gates are used for both trainable processing and data encoding. When entanglement is introduced, Controlled-Z (CZ) gates are applied between qubit pairs.

The parameter set for each circuit is divided into two categories:

- θ , the base rotation angles, organized as a tensor of shape (qubits, layers, 3),
- α , the data encoding weights, shaped as (qubits, layers, data dimension).

The total number of trainable parameters scales with both the number of qubits and the number of re-uploading layers. For example, the single-qubit configuration contains $3 \times \text{layers}$ trainable parameters. The two-qubit configuration without entanglement uses two parallel $U(\phi)$ gates per layer (one on each qubit), resulting in $6 \times \text{layers}$ parameters. When entanglement is included, the same number of $U(\phi)$ gates are used, along with $(\text{layers}-1)$ Controlled-Z gates placed between adjacent qubit layers.

The data encoding follows the transformation $\theta_{\text{encoded}} = \theta + \alpha \otimes x$, where x is the input feature vector. This allows the same circuit structure to dynamically adapt to different input data points while preserving trainable components.

Class label encoding differs based on the cost function used. For fidelity-based classification, labels are represented as computational basis states such as $|0\rangle$ or $|1\rangle$. For trace-distance-based classification, target class states are defined using Bloch sphere coordinates.

Supplementary Note 7: Optimization Methods

In practice, deploying a parameterized quantum classifier involves a process of minimizing within the parameter space that delineates the circuit's configuration. The process is often termed a hybrid algorithm, denoting the symbiotic relationship and advantages derived from combining quantum logic and classical logic. In particular, the ensemble of angles (θ_i) and weights (w_i) defines a parameter space that requires systematic exploration to achieve the minimization of χ^2 .

The occurrence of local minima is unavoidable. The arrangement of rotation gates results in an intricate multiplication of independent trigonometric functions, suggesting that our problem is characterized by a widespread distribution of minima.

The primary challenge boils down to minimizing a function that is defined by a vast array of parameters. In the case of a single-qubit classifier, the total number of parameters can be expressed as, where represents the problem's dimension (that is, the dimension of), and signifies the number of layers. Among these parameters, three are rotational angles, while the rest pertain to the weight [1]. To identify the most effective solution, we evaluate the performance of four distinct minimization techniques: the L-BFGS-B method, the COBYLA method, the Nelder-Mead method, and the Sequential Least Squares Programming (SLSQP) method.

The key challenge in optimizing a single-qubit classifier involves minimizing a function across a complex parameter space, calculated as $(3 + d)N$, where "d" is the problem's dimension and "N" is the number of layers. Also, in addition,

we need to consider rotational angles and the weight (\bar{w}_i) corresponding to the dimension ¹. To discover the optimal solution, we delve into the efficiency of four diverse minimization strategies: the L-BFGS-B, COBYLA, Nelder-Mead, and Sequential Least Squares Programming (SLSQP) methods.

A. L-BFGS-B METHOD

The L-BFGS-B technique, part of the quasi-Newton optimization methods, refines the Broyden–Fletcher–Goldfarb–Shanno (BFGS) approach by efficiently using limited computer memory ¹⁰. Its design excels in handling optimization tasks involving numerous variables, offering a linear memory usage advantage, making it highly effective for large-scale problems ¹¹.

The L-BFGS-B method is widely recognized as a cornerstone technique across various advanced applications in the field of graphics ^{12,13}. It specializes in minimizing a scalar function of one or several variables by initiating with a preliminary estimate of the optimum value. Through iterative refinement, it progressively improves upon this initial estimate to approach an optimal solution. The method employs function derivatives to determine the direction of steepest descent and approximates the Hessian matrix (second-order derivatives) using limited memory. The parameter update rule is given by¹⁴:

$$\theta_{k+1} = \theta_k - \alpha_k H_k^{-1} \nabla f(\theta_k)$$

where θ_k is the current parameter vector, $\nabla f(\theta_k)$ is the gradient, H_k^{-1} is an approximation of the inverse Hessian, and α_k is a step size typically determined by line search. This method is particularly efficient in handling large-scale problems due to its low memory usage and fast matrix-vector multiplications.

B. CONSTRAINED OPTIMIZATION BY LINEAR APPROXIMATION METHOD

COBYLA (Constrained Optimization BY Linear Approximation) is an optimization algorithm designed to minimize a scalar objective function that depends on one or more variables, subject to constraints ^{15,16}. One of the key features of COBYLA is that it does not require the calculation of derivatives, such as gradients or Hessians, of the objective function or constraints. This makes COBYLA particularly useful in situations where the derivatives are unknown, unreliable, or computationally expensive to obtain ¹⁵. Instead of requiring gradients or Hessians, COBYLA constructs linear approximations of both the objective function and constraints within a trust region framework. At each iteration, it solves a subproblem defined by: $\min_{\theta} f(\theta)$ subject to $c_i(\theta) \geq 0$ and approximates the objective function locally as:

$$f(\theta + \Delta\theta) \approx f(\theta) + \nabla f(\theta)^T \Delta\theta$$

although $\nabla f(\theta)$ is never explicitly calculated—its effect is estimated using linear interpolation.

COBYLA has been effectively utilized in quantum computing, especially as a classical optimization routine within Variational Hybrid Quantum-Classical Algorithms (VHQCs) ¹⁷. These algorithms employ a parameterized quantum circuit, or ansatz, which is refined through a dynamic interchange between a classical computer and a quantum device. The classical computer adjusts the ansatz's parameters to minimize a cost function, which the quantum device efficiently evaluates. Through iterative updates based on the cost function outcomes, the VHQCA aims to discover the most effective ansatz configuration for specific problems. The derivative-free characteristic of COBYLA makes it particularly advantageous for this setting, where the cost functions often lack easily computable or analytically defined derivatives.

C. NELDER-MEAD METHOD

The Nelder-Mead algorithm, introduced by John Nelder and Roger Mead in 1965, is a widely used direct search method for unconstrained optimization problems ¹⁸. The algorithm operates by maintaining a simplex of $n+1$ points in an n -dimensional space, iteratively moving the simplex toward the optimal solution through a series of transformations, including reflection, expansion, contraction, and shrinkage ¹⁸. These operations are defined as follows:

- **Reflection:**

$$\theta_r = \bar{\theta} + \alpha(\bar{\theta} - \theta_h)$$

- **Expansion:**

$$\theta_e = \bar{\theta} + \gamma(\theta_r - \bar{\theta})$$

- **Contraction:**

$$\theta_c = \bar{\theta} + \rho(\theta_r - \bar{\theta})$$

- **Shrinkage:**

$$\theta_i = \theta_l + \sigma(\theta_i - \theta_l)$$

Here, $\bar{\theta}$ is the centroid of the best n points, θ_h is the worst-performing point, and α, γ, ρ , and σ are user-defined coefficients controlling the behavior of each transformation. This method is especially effective in low-dimensional, non-

convex optimization landscapes and is widely used when the objective function is noisy, non-differentiable, or discontinuous.

Recent studies have focused on enhancing the Nelder-Mead algorithm to improve its efficiency and adaptability. Gao and Han¹⁹ proposed an implementation of the Nelder-Mead algorithm with adaptive parameters, which can automatically adjust the parameter values based on the optimization progress. This adaptive approach has been shown to improve the algorithm's convergence speed and solution quality¹⁹.

Its capacity to address problems in which derivative information is not readily accessible renders it a favorable option for numerous applications in QML. However, it is essential to conduct comprehensive evaluations to scrutinize the method's accuracy, efficiency, and sensitivity to the initial guess for each unique application^{20,21}.

D. SEQUENTIAL LEAST SQUARES PROGRAMMING METHOD

The Sequential Least Squares Programming (SLSQP) method is an optimization technique that minimizes functions while adhering to specific constraints²². It is based on Sequential Quadratic Programming (SQP), which simplifies the optimization problem into a series of smaller, more manageable quadratic subproblems. In each subproblem, a quadratic approximation of the objective function and constraints is constructed using a second-order parabolic curve to model the function's behavior near a specific point. SLSQP updates this approximation using the quasi-Newton method. Specifically, the subproblem it solves takes the form:

$$\min_{\Delta\theta} \quad \left[\frac{1}{2} \Delta\theta^T B_k \Delta\theta + \nabla f(\theta_k)^T \Delta\theta \right]$$

subject to:

$$\begin{aligned} c_i(\theta_k) + \nabla c_i(\theta_k)^T \Delta\theta &\geq 0 \text{ (inequality constraints)} \\ h_j(\theta_k) + \nabla h_j(\theta_k)^T \Delta\theta &= 0 \text{ (equality constraints)} \end{aligned}$$

where B_k is an approximation to the Hessian of the Lagrangian, and ∇f , ∇c_i , and ∇h_j are gradients of the objective and constraint functions.

Additionally, SLSQP applies a least-squares method to solve these quadratic subproblems, striving to minimize the total squared deviations between the approximation and actual function values. This method can handle both equality and inequality constraints, including variable bounds, by integrating a penalty function that imposes additional costs for any constraint or bound violations. SLSQP ensures efficient convergence by terminating the optimization process upon meeting a predefined convergence criterion, typically related to changes in the objective function value or the gradient vector's norm. This safeguard prevents indefinite computations, ensuring timely solutions.

Local minima are common challenges in both neural networks and quantum classifiers due to their complex mathematical structures—neural networks with compounded nonlinear functions and quantum circuits with prevalent trigonometric functions. This complexity increases the likelihood of encountering local minima during optimization. Moreover, with smaller training sets, the choice of optimization method is crucial. For instance, the Nelder-Mead method is noted for its robustness, particularly its lower susceptibility to local minima.

It is also critical to recognize that minimization methods are sensitive to noise, which can significantly impact their effectiveness, especially in practical quantum computing applications¹⁷.

Supplementary Note 8: Comparing the developed code for this research with original reference.

# coding=utf-8	=====	
# Adrian Pérez-Salinas, Alba Cervera-Lierta, Elies Gil, J. Ignacio Latorre	<=	# Adrian Pérez-Salinas, Alba Cervera-Lierta, Elies Gil, J. Ignacio Latorre
# Sara Aminiour, Mike Banad, Sarah Sharif	<=	# Sara Aminiour, Mike Banad, Sarah Sharif
# September 25th 2024	<=	# June 3rd 2025
#School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73819 USA.	<=	#Universitat de Barcelona / Barcelona Supercomputing Center/Institut de Ciències del Cosmos
#####	<=	#####
10 #The code on the left was developed by Sara Aminiour, while the code on the right serves as the reference implementation	<=	10 #The code on the left was developed by Sara Aminiour, while the code on the right serves as the reference implementation
11 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.	<=	11 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.
12 Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem as well as necessary modification to apply CQVLA, LBFGS-B, WELDER-HEAD, and SLSD minimization methods.	<=	12 #This file is a file taking many different functions from other files and mixing them all together
13 #####		
14 # so that the usage is automated	=	14 # so that the usage is automated
15 from data_gen import data_generator	=	15 from data_gen import data_generator
16 from problem_gen import problem_generator, representatives, representatives_tr	=	16 from problem_gen import problem_generator, representatives
17 from fidelity_minimization import fidelity_minimization	=	17 from fidelity_minimization import fidelity_minimization
18 from trace_minimization import trace_minimization	=	18 from trace_minimization import trace_minimization
19 from test_data import Accuracy_test, tester	=	19 from test_data import Accuracy_test, tester
20 from save_data import write_summary, read_summary, name_folder, samples_paint, samples_paint_worldmap, laea_x, laea_y, write_epochs_error_rate	=	20 from save_data import write_summary, read_summary, name_folder, samples_paint, samples_paint_worldmap, laea_x, laea_y, write_epochs_error_rate
21 from save_data import write_epochs_file, write_epoch, close_epochs_file, create_folder, write_epochs_error_rate	=	21 from save_data import write_epochs_file, write_epoch, close_epochs_file, create_folder, write_epochs_error_rate
22 import numpy as np	=	22 import numpy as np
23 from matplotlib.pyplot import plt	=	23 from matplotlib.pyplot import plt
24 from circuitry import code_coords, circuit	=	24 from circuitry import code_coords, circuit
25 from matplotlib import import get_cmap	=	25 from matplotlib import import get_cmap
26 from matplotlib.colors import Normalize	=	26 from matplotlib.colors import Normalize
27 def minimizer(chi, problem, qubits, entanglement, layers, method, name, epochs=3000, batch_size=20, eta=0.1):	<=	27 def minimizer(chi, problem, qubits, entanglement, layers, method, name, epochs=3000, batch_size=20, eta=0.1):
28 ...	=	28 ...
29 This function creates data and minimizes whichever problem (from the selected ones)	=	29 This function creates data and minimizes whichever problem (from the selected ones)
30 INPUT:	=	30 INPUT:
31 -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'	=	31 -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
32 -problem: name of the problem, to choose among	=	32 -problem: name of the problem, to choose among
33 ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy	=	33 ['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy
34 lines']	=	34 'squares', 'wavy lines']
35 -qubits: number of qubits, must be an integer	=	35 -qubits: number of qubits, must be an integer
36 -entanglement: whether there is entanglement or not in the Ansatz, just 'y'/n'	=	36 -entanglement: whether there is entanglement or not in the Ansatz, just 'y'/n'
37 -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account	=	37 -layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
38 -method: minimization method, to choose among ['SGD', another valid for function scipy.optimize.minimize]	=	38 -method: minimization method, to choose among ['SGD', another valid for function
39 -name: a name we want for our our files to be save with	=	39 -name: a name we want for our our files to be save with
40 -seed: seed of numpy.random, needed for replicating results	=	40 -seed: seed of numpy.random, needed for replicating results
41 -epochs: number of epochs for a 'SGD' method. If there is another method, this input has got no importance	=	41 -epochs: number of epochs for a 'SGD' method. If there is another method, this input has got no importance
42 -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method	=	42 -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method
43 -eta: Learning rate, only for 'SGD' method	=	43 -eta: Learning rate, only for 'SGD' method
44 OUTPUT:	=	44 OUTPUT:
45 This function has got no outputs, but several files are saved in an appropriate folder. The files are	=	45 This function has got no outputs, but several files are saved in an appropriate folder. The files are
46 -summary.txt: Saves useful information for the problem	=	46 -summary.txt: Saves useful information for the problem
47 -theta.txt: saves the theta parameters as a flat array	=	47 -theta.txt: saves the theta parameters as a flat array
48 -alpha.txt: saves the alpha parameters as a flat array	=	48 -alpha.txt: saves the alpha parameters as a flat array
49 -weight.txt: saves the weights as a flat array if they exist	=	49 -weight.txt: saves the weights as a flat array if they exist
50 ...	=	50 ...
51 data, drawing = data_generator(problem)	=	51 no.random.seed(seed)
52 if problem == 'sphere':	=	52 if problem == 'sphere':
53 train_data = data[500]	=	53 train_data = data[500]
54 elif problem == 'hypersphere':	=	54 test_data = data[500]
55 train_data = data[1000]	=	55 train_data = data[1000]
56 test_data = data[1000]	=	56 test_data = data[1000]
57 else:	=	57 else:
58 train_data = data[250]	=	58 train_data = data[200]
59 test_data = data[250]	=	59 test_data = data[200]
60	=	60
61 if chi == 'fidelity_chi':	=	61 if chi == 'fidelity_chi':
62 acc_train=0	=	62 acc_train=0
63 Accuracy_test=0	=	63 Accuracy_test=0
64 i=1	=	64 while i<=1:
65 qubits_lab = qubits	=	65 qubits_lab = qubits
66 theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,	=	66 theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,
67 qubits_lab=qubits_lab)	=	67 qubits_lab=qubits_lab)
68 theta, alpha, f = fidelity_minimization(theta, alpha, train_data, reprs,	=	68 theta, alpha, f = fidelity_minimization(theta, alpha, train_data, reprs,
69 entanglement, method,	=	69 entanglement, method,
70 batch_size, eta, epochs)	=	70 batch_size, eta, epochs)
71	=	71
72 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)	=	72 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
73 Accuracy_tr=acc_train	=	73 Accuracy_tr=acc_train
74 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)	=	74 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
75 Accuracy_tevacc_test	=	75 Accuracy_tevacc_test
76 text_file_nm = open('acc.txt', mode='a')	=	76 text_file_nm = open('acc.txt', mode='a')
77 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	77 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
78 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	78 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
79 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	79 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
80 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	80 text_file_nm.write('===== =====') 81 text_file_nm.close()
81 text_file_nm.write('===== =====')	=	81 text_file_nm.close()
82 text_file_nm.close()	=	82 text_file_nm.close()
83	=	83
84 i=1	=	84 i=1
85 print(i-1)	=	85 print(i-1)
86 atr=Accuracy_tr(i-1)	=	86 atr=Accuracy_tr(i-1)
87 ate=Accuracy_te(i-1)	=	87 ate=Accuracy_te(i-1)
88	=	88
89 text_file_nm = open('AverageAcc.txt', mode='a')	=	89 text_file_nm = open('AverageAcc.txt', mode='a')
90 text_file_nm.write(problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	90 text_file_nm.write(problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
91 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	91 text_file_nm.write('===== =====') 92 text_file_nm.close()
92 text_file_nm.write('===== =====')	=	92 text_file_nm.close()
93 text_file_nm.close()	=	93 text_file_nm.close()
94	=	94
95 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	95 write_summary(chi, problem, qubits, entanglement, layers, method, name,
96 theta, alpha, 0, f, atr, ate, epochs=epochs)	=	96 theta, alpha, 0, f, atr, ate, epochs=epochs)
97 elif chi == 'trace_chi':	=	97 elif chi == 'trace_chi':
98 Accuracy_tr=0	=	98 Accuracy_tr=0
99 Accuracy_test=0	=	99 Accuracy_test=0
100 i=1	=	100 while i<=1:
101 qubits_lab = qubits	=	101 qubits_lab = qubits
102 theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,	=	102 theta, alpha, reprs = problem_generator(problem,qubits, layers, chi,
103 qubits_lab=qubits_lab)	=	103 qubits_lab=qubits_lab)
104 theta, alpha, f = trace_minimization(theta, alpha, train_data, reprs,	=	104 theta, alpha, f = trace_minimization(theta, alpha, train_data, reprs,
105 entanglement, method,	=	105 entanglement, method,
106 batch_size, eta, epochs)	=	106 batch_size, eta, epochs)
107	=	107
108 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)	=	108 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi)
109 Accuracy_tr=acc_train	=	109 Accuracy_tr=acc_train
110 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)	=	110 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi)
111 Accuracy_tevacc_test	=	111 Accuracy_tevacc_test
112 text_file_nm = open('acc.txt', mode='a')	=	112 text_file_nm = open('acc.txt', mode='a')
113 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	113 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
114 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	114 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
115 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	115 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
116 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	116 text_file_nm.write('===== =====') 117 text_file_nm.close()
117 text_file_nm.write('===== =====')	=	117 text_file_nm.close()
118 text_file_nm.close()	=	118 text_file_nm.close()
119	=	119
120 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	120 write_summary(chi, problem, qubits, entanglement, layers, method, name,
121 theta, alpha, 0, f, atr, ate, epochs=epochs)	=	121 theta, alpha, 0, f, atr, ate, epochs=epochs)
122 elif chi == 'weighted_fidelity_chi':	=	122 elif chi == 'weighted_fidelity_chi':
123 Accuracy_tr=0	=	123 Accuracy_tr=0
124 Accuracy_test=0	=	124 Accuracy_test=0
125 i=1	=	125 while i<=1:
126 qubits_lab = 1	=	126 qubits_lab = 1
127 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,	=	127 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
128 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,	=	128 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
129 entanglement, method)	=	129 entanglement, method)
130	=	130
131 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)	=	131 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)
132 Accuracy_tr=acc_train	=	132 Accuracy_tr=acc_train
133 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)	=	133 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)
134 Accuracy_tevacc_test	=	134 Accuracy_tevacc_test
135 text_file_nm = open('acc.txt', mode='a')	=	135 text_file_nm = open('acc.txt', mode='a')
136 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	136 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
137 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	137 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
138 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	138 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
139 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	139 text_file_nm.write('===== =====') 140 text_file_nm.close()
140 text_file_nm.write('===== =====')	=	140 text_file_nm.close()
141 text_file_nm.close()	=	141 text_file_nm.close()
142	=	142
143 i=1	=	143 i=1
144 print(i-1)	=	144 print(i-1)
145 atr=Accuracy_tr(i-1)	=	145 atr=Accuracy_tr(i-1)
146 ate=Accuracy_te(i-1)	=	146 ate=Accuracy_te(i-1)
147	=	147
148 text_file_nm = open('AverageAcc.txt', mode='a')	=	148 text_file_nm = open('AverageAcc.txt', mode='a')
149 text_file_nm.write(problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	149 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
150 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	150 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
151 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	151 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
152 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	152 text_file_nm.write('===== =====') 153 text_file_nm.close()
153 text_file_nm.write('===== =====')	=	153 text_file_nm.close()
154 text_file_nm.close()	=	154 text_file_nm.close()
155	=	155
156 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	156 write_summary(chi, problem, qubits, entanglement, layers, method, name,
157 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)	=	157 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)
158 elif chi == 'weighted_fidelity_chi':	=	158 elif chi == 'weighted_fidelity_chi':
159 Accuracy_tr=0	=	159 Accuracy_tr=0
160 Accuracy_test=0	=	160 Accuracy_test=0
161 i=1	=	161 while i<=1:
162 qubits_lab = 1	=	162 qubits_lab = 1
163 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,	=	163 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
164 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,	=	164 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
165 entanglement, method)	=	165 entanglement, method)
166	=	166
167 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)	=	167 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)
168 Accuracy_tr=acc_train	=	168 Accuracy_tr=acc_train
169 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)	=	169 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)
170 Accuracy_tevacc_test	=	170 Accuracy_tevacc_test
171 text_file_nm = open('acc.txt', mode='a')	=	171 text_file_nm = open('acc.txt', mode='a')
172 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	172 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
173 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	173 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
174 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	174 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
175 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	175 text_file_nm.write('===== =====') 176 text_file_nm.close()
176 text_file_nm.write('===== =====')	=	176 text_file_nm.close()
177 text_file_nm.close()	=	177 text_file_nm.close()
178	=	178
179 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	179 write_summary(chi, problem, qubits, entanglement, layers, method, name,
180 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)	=	180 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)
181 elif chi == 'weighted_fidelity_chi':	=	181 elif chi == 'weighted_fidelity_chi':
182 Accuracy_tr=0	=	182 Accuracy_tr=0
183 Accuracy_test=0	=	183 Accuracy_test=0
184 i=1	=	184 while i<=1:
185 qubits_lab = 1	=	185 qubits_lab = 1
186 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,	=	186 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
187 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,	=	187 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
188 entanglement, method)	=	188 entanglement, method)
189	=	189
190 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)	=	190 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)
191 Accuracy_tr=acc_train	=	191 Accuracy_tr=acc_train
192 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)	=	192 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)
193 Accuracy_tevacc_test	=	193 Accuracy_tevacc_test
194 text_file_nm = open('acc.txt', mode='a')	=	194 text_file_nm = open('acc.txt', mode='a')
195 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	195 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
196 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	196 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
197 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	197 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
198 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	198 text_file_nm.write('===== =====') 199 text_file_nm.close()
199 text_file_nm.write('===== =====')	=	199 text_file_nm.close()
200 text_file_nm.close()	=	200 text_file_nm.close()
201	=	201
202 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	202 write_summary(chi, problem, qubits, entanglement, layers, method, name,
203 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)	=	203 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)
204 elif chi == 'weighted_fidelity_chi':	=	204 elif chi == 'weighted_fidelity_chi':
205 Accuracy_tr=0	=	205 Accuracy_tr=0
206 Accuracy_test=0	=	206 Accuracy_test=0
207 i=1	=	207 while i<=1:
208 qubits_lab = 1	=	208 qubits_lab = 1
209 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,	=	209 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
210 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,	=	210 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
211 entanglement, method)	=	211 entanglement, method)
212	=	212
213 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)	=	213 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)
214 Accuracy_tr=acc_train	=	214 Accuracy_tr=acc_train
215 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)	=	215 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)
216 Accuracy_tevacc_test	=	216 Accuracy_tevacc_test
217 text_file_nm = open('acc.txt', mode='a')	=	217 text_file_nm = open('acc.txt', mode='a')
218 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	218 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
219 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	219 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
220 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	220 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
221 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	221 text_file_nm.write('===== =====') 222 text_file_nm.close()
222 text_file_nm.write('===== =====')	=	222 text_file_nm.close()
223 text_file_nm.close()	=	223 text_file_nm.close()
224	=	224
225 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	225 write_summary(chi, problem, qubits, entanglement, layers, method, name,
226 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)	=	226 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)
227 elif chi == 'weighted_fidelity_chi':	=	227 elif chi == 'weighted_fidelity_chi':
228 Accuracy_tr=0	=	228 Accuracy_tr=0
229 Accuracy_test=0	=	229 Accuracy_test=0
230 i=1	=	230 while i<=1:
231 qubits_lab = 1	=	231 qubits_lab = 1
232 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,	=	232 theta, alpha, weight, reprs = problem_generator(problem,qubits, layers, chi,
233 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,	=	233 theta, alpha, weight, f = weighted_fidelity_minimization(theta, alpha, weight, train_data, reprs,
234 entanglement, method)	=	234 entanglement, method)
235	=	235
236 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)	=	236 acc_train = tester(theta, alpha, train_data, reprs, entanglement, chi, weights=weight)
237 Accuracy_tr=acc_train	=	237 Accuracy_tr=acc_train
238 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)	=	238 acc_test = tester(theta, alpha, test_data, reprs, entanglement, chi, weights=weight)
239 Accuracy_tevacc_test	=	239 Accuracy_tevacc_test
240 text_file_nm = open('acc.txt', mode='a')	=	240 text_file_nm = open('acc.txt', mode='a')
241 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	241 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)
242 entanglement + ' + method + ' + acc_train + ' + ' + str(acc_train)	=	242 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +
243 text_file_nm.write(str(i) + problem + ' + chi + ' + str(qubits) + 'Qubits' + str(layers) + 'Layers' +	=	243 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)
244 entanglement + ' + method + ' + acc_test + ' + ' + str(acc_test)	=	244 text_file_nm.write('===== =====') 245 text_file_nm.close()
245 text_file_nm.write('===== =====')	=	245 text_file_nm.close()
246 text_file_nm.close()	=	246 text_file_nm.close()
247	=	247
248 write_summary(chi, problem, qubits, entanglement, layers, method, name,	=	248 write_summary(chi, problem, qubits, entanglement, layers, method, name,
249 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)	=	249 theta, alpha, 0, f, acc_train, acc_test, seed, epochs=epochs)
250 elif chi == 'weighted_fidelity_chi':	=	250 elif chi == 'weighted_fidelity_chi':
251 Accuracy_tr=0	=	251 Accuracy_tr=0
252 Accuracy_test=0	=	252 Accuracy_test=0
253 i=1	=	253 while i<=1:
254 qubits_lab =		

1	# coding=utf-8	+~	
2	#####	=	1 #####
3	#Quantum classifier		2 #Quantum classifier
4	#Sara Aminpour, Mike Banad, Sarah Sharif	<>	3 #Adrián Pérez-Salinas, Alba Cervera-Liarta, Elies Gil, J. Ignacio Latorre
5	#September 25th 2024		4 #Code by APS
			5 #Code checks by ACL
6			6 #June 3rd 2019
7	#School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,	<>	7
8	#####		8
9	#IMPORTANT NOTE:		
10	#The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference		
11	implementation by Adrián Pérez-Salinas.		
12	#The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.		
13	#Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem		9 #Universitat de Barcelona / Barcelona Supercomputing Center/Institut de Ciències del Cosmos
14	#as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.		10
15	#####	=	11 #####
16	## This file creates the data points for the different problems to be tackled by the quantum classifier		12 ## This file creates the data points for the different problems to be tackled by the quantum classifier
17			13
18			14
19			15
20	import numpy as np		16 import numpy as np
21			17
22	problems = ['circle', 'line', '3 circles', 'wavy circle', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines']	<>	19 problems = ['circle', '3 circles', 'wavy circle', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines']
23		=	20
24	def data_generator(problem, samples=None):		21 def data_generator(problem, samples=None):
25	"""		22 """
26	This function generates the data for a problem		23 This function generates the data for a problem
27	INPUT:		24 INPUT:
28	-problem: Name of the problem, one of: 'circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines'		25 -problem: Name of the problem, one of: 'circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines'
29	-samples: Number of samples for the data		26 -samples: Number of samples for the data
30	OUTPUT:		27 OUTPUT:
31	-data: set of training and test data		28 -data: set of training and test data
32	-settings: things needed for drawing		29 -settings: things needed for drawing
33	"""		30 """
34	problem = problem.lower()		31 problem = problem.lower()
35	if problem not in problems:		32 if problem not in problems:
36	raise ValueError('problem must be one of {}'.format(problems))		33 raise ValueError('problem must be one of {}'.format(problems))
37	if samples == None:		34 if samples == None:
38	if problem == 'sphere':		35 if problem == 'sphere':
39	samples = 4500		36 samples = 4500
40	elif problem == 'hypersphere':		37 elif problem == 'hypersphere':
41	samples = 5000		38 samples = 5000
42	else:		39 else:
43	samples = 4250	<>	40 samples = 4200
44		=	41
45	if problem == 'circle':		42 if problem == 'circle':
46	data, settings = _circle(samples)		43 data, settings = _circle(samples)
47			44
48	if problem == '3 circles':		45 if problem == '3 circles':
49	data, settings = _3_circles(samples)		46 data, settings = _3_circles(samples)
50			47
51	if problem == 'wavy lines':		48 if problem == 'wavy lines':
52	data, settings = _wavy_lines(samples)		49 data, settings = _wavy_lines(samples)
53			50
54	if problem == 'squares':		51 if problem == 'squares':
55	data, settings = _squares(samples)		52 data, settings = _squares(samples)
56			53
57	if problem == 'sphere':		54 if problem == 'sphere':
58	data, settings = _sphere(samples)		55 data, settings = _sphere(samples)
59			56
60	if problem == 'non convex':		57 if problem == 'non convex':
61	data, settings = _non_convex(samples)		58 data, settings = _non_convex(samples)
62			59
63	if problem == 'crown':		60 if problem == 'crown':
64	data, settings = _crown(samples)		61 data, settings = _crown(samples)
65			62
66	if problem == 'tricrown':		63 if problem == 'tricrown':
67	data, settings = _tricrown(samples)		64 data, settings = _tricrown(samples)
68			65
69	if problem == 'hypersphere':		66 if problem == 'hypersphere':
70	data, settings = _hypersphere(samples)		67 data, settings = _hypersphere(samples)
71	#####	<>	
72	if problem == 'line':		
73	data, settings = _line(samples)		
74	#####		
75			68
			69
76	return data, settings	=	70 return data, settings
77			71
78	def _circle(samples):		72 def _circle(samples):
79	centers = np.array([[0, 0]])		73 centers = np.array([[0, 0]])
80	radii = np.array([np.sqrt(2/np.pi)])		74 radii = np.array([np.sqrt(2/np.pi)])
81	data=[]		75 data=[]
82	dim = 2		76 dim = 2
83	for i in range(samples):		77 for i in range(samples):
84	x = 2 * (np.random.rand(dim)) - 1		78 x = 2 * (np.random.rand(dim)) - 1
85	y = 0		79 y = 0
86	for c, r in zip(centers, radii):		80 for c, r in zip(centers, radii):
87	if np.linalg.norm(x - c) < r:		81 if np.linalg.norm(x - c) < r:
88	y = 1		82 y = 1
89			83
90	data.append([x, y])		84 data.append([x, y])
91		<>	85
92	return data, (centers, radii)	=	86 return data, (centers, radii)
93			87
94	def _3_circles(samples):		88 def _3_circles(samples):
95	centers = np.array([[-1, 1], [1, 0], [-.5, -.5]])		89 centers = np.array([[-1, 1], [1, 0], [-.5, -.5]])
96	radii = np.array([1, np.sqrt(6/np.pi - 1), 1/2])		90 radii = np.array([1, np.sqrt(6/np.pi - 1), 1/2])
97	data=[]		91 data=[]
98	dim = 2		92 dim = 2
99	for i in range(samples):		93 for i in range(samples):
100	x = 2 * (np.random.rand(dim)) - 1		94 x = 2 * (np.random.rand(dim)) - 1
101	y = 0		95 y = 0
102	for j, (c, r) in enumerate(zip(centers, radii)):		96 for j, (c, r) in enumerate(zip(centers, radii)):
103	if np.linalg.norm(x - c) < r:		97 if np.linalg.norm(x - c) < r:
104	y = j + 1		98 y = j + 1
105			99
106	data.append([x, y])		100 data.append([x, y])
107			101
108	return data, (centers, radii)		102 return data, (centers, radii)
109			103
110			104
111			105
112	def _wavy_lines(samples, freq = 1):		106 def _wavy_lines(samples, freq = 1):
113	def fun1(s):		107 def fun1(s):
114	return s + np.sin(freq * np.pi * s)		108 return s + np.sin(freq * np.pi * s)
115			109
116	def fun2(s):		110 def fun2(s):
117	return -s + np.sin(freq * np.pi * s)		111 return -s + np.sin(freq * np.pi * s)
118	data=[]		112 data=[]
119	dim=2		113 dim=2
120	for i in range(samples):		114 for i in range(samples):
121	x = 2 * (np.random.rand(dim)) - 1		115 x = 2 * (np.random.rand(dim)) - 1
122	if x[1] < fun1(x[0]) and x[1] < fun2(x[0]): y = 0		116 if x[1] < fun1(x[0]) and x[1] < fun2(x[0]): y = 0
123	if x[1] < fun1(x[0]) and x[1] > fun2(x[0]): y = 1		117 if x[1] < fun1(x[0]) and x[1] > fun2(x[0]): y = 1
124	if x[1] > fun1(x[0]) and x[1] < fun2(x[0]): y = 2		118 if x[1] > fun1(x[0]) and x[1] < fun2(x[0]): y = 2
125	if x[1] > fun1(x[0]) and x[1] > fun2(x[0]): y = 3		119 if x[1] > fun1(x[0]) and x[1] > fun2(x[0]): y = 3
126	data.append([x, y])		120 data.append([x, y])
127			121
128	return data, freq		122 return data, freq
129			123
130	def _squares(samples):		124 def _squares(samples):
131	data=[]		125 data=[]
132	dim=2		126 dim=2
133	for i in range(samples):		127 for i in range(samples):
134	x = 2 * (np.random.rand(dim)) - 1		128 x = 2 * (np.random.rand(dim)) - 1
135	if x[0] < 0 and x[1] < 0: y = 0		129 if x[0] < 0 and x[1] < 0: y = 0
136	if x[0] < 0 and x[1] > 0: y = 1		130 if x[0] < 0 and x[1] > 0: y = 1
137	if x[0] > 0 and x[1] < 0: y = 2		131 if x[0] > 0 and x[1] < 0: y = 2
138	if x[0] > 0 and x[1] > 0: y = 3		132 if x[0] > 0 and x[1] > 0: y = 3
139	data.append([x, y])		133 data.append([x, y])
140			134
141	return data, None		135 return data, None
142			136
143	#####	+~	
144	def _line(samples):		
145	data=[]		
146	dim=2		
147	for i in range(samples):		
148	x = 2 * np.random.rand(dim) -1		
149	#x = np.random.rand(dim)		
150	if x[0] < x[1] : y = 0		
151	if x[0] > x[1] : y = 1		
152			
153	data.append([x, y])		
154			
155	return data, None		
156	#####		
157		=	137
158	def _non_convex(samples, freq = 1, x_val = 2, sin_val = 1.5):		138 def _non_convex(samples, freq = 1, x_val = 2, sin_val = 1.5):
159	def fun(s):		139 def fun(s):
160	return -x_val * s + sin_val * np.sin(freq * np.pi * s)		140 return -x_val * s + sin_val * np.sin(freq * np.pi * s)
161			141
162	data = []		142 data = []
163	dim = 2		143 dim = 2
164	for i in range(samples):		144 for i in range(samples):
165	x = 2 * (np.random.rand(dim)) - 1		145 x = 2 * (np.random.rand(dim)) - 1
166	if x[1] < fun(x[0]): y = 0		146 if x[1] < fun(x[0]): y = 0
167	if x[1] > fun(x[0]): y = 1		147 if x[1] > fun(x[0]): y = 1
168	data.append([x, y])		148 data.append([x, y])
169			149
170	return data, (freq, x_val, sin_val)		150 return data, (freq, x_val, sin_val)
171			151
172	def _crown(samples):		152 def _crown(samples):
173	c = [[0,0],[0,0]]		153 c = [[0,0],[0,0]]
174	r = [np.sqrt(.8), np.sqrt(.8 - 2/np.pi)]		154 r = [np.sqrt(.8), np.sqrt(.8 - 2/np.pi)]
175	data = []		155 data = []
176	dim = 2		156 dim = 2
177	for i in range(samples):		157 for i in range(samples):
178	x = 2 * (np.random.rand(dim)) - 1		158 x = 2 * (np.random.rand(dim)) - 1
179	if np.linalg.norm(x - c[0]) < r[0] and np.linalg.norm(x - c[1]) > r[1]:		159 if np.linalg.norm(x - c[0]) < r[0] and np.linalg.norm(x - c[1]) > r[1]:
180	y = 1		160 y = 1
181	else:		161 else:
182	y=0		162 y=0
183	data.append([x, y])		163 data.append([x, y])
184			164
185	return data, (c, r)		165 return data, (c, r)
186			166
187			167
188			168
189	def _tricrown(samples):		169 def _tricrown(samples):
190	Centers = [[0,0],[0,0]]		170 Centers = [[0,0],[0,0]]
191	radii = [np.sqrt(.8 - 2/np.pi), np.sqrt(.8)]		171 radii = [np.sqrt(.8 - 2/np.pi), np.sqrt(.8)]
192	data = []		172 data = []
193	dim = 2		173 dim = 2
194	for i in range(samples):		174 for i in range(samples):
195	x = 2 * (np.random.rand(dim)) - 1		175 x = 2 * (np.random.rand(dim)) - 1
196	y=0		176 y=0
197	for j,(r,c) in enumerate(zip(radii, centers)):		177 for j,(r,c) in enumerate(zip(radii, centers)):
198	if np.linalg.norm(x - c) > r:		178 if np.linalg.norm(x - c) > r:
199	y = j + 1		179 y = j + 1
200	data.append([x, y])		180 data.append([x, y])
201			181
202	return data, (centers, radii)		182 return data, (centers, radii)
203			183
204	def _sphere(samples):		184 def _sphere(samples):
205	Centers = np.array([[0, 0, 0]])		185 Centers = np.array([[0, 0, 0]])
206	radii = np.array([(3/np.pi)**(1/3)])		186 radii = np.array([(3/np.pi)**(1/3)])
207	data=[]		187 data=[]
208	dim = 3		188 dim = 3
209	for i in range(samples):		189 for i in range(samples):
210	x = 2 * (np.random.rand(dim)) - 1		190 x = 2 * (np.random.rand(dim)) - 1
211	y = 0		191 y = 0
212	for c, r in zip(centers, radii):		192 for c, r in zip(centers, radii):
213	if np.linalg.norm(x - c) < r:		193 if np.linalg.norm(x - c) < r:
214	y = 1		194 y = 1
215			195
216	data.append([x, y])		196 data.append([x, y])
217			197
218	return data, (centers, radii)		198 return data, (centers, radii)
219			199
220	def _hypersphere(samples):		200 def _hypersphere(samples):
221	Centers = np.array([[0, 0, 0, 0]])		201 Centers = np.array([[0, 0, 0, 0]])
222	radii = np.array([(2/np.pi)**(1/2)])		202 radii = np.array([(2/np.pi)**(1/2)])
223	data=[]		203 data=[]
224	dim = 4		204 dim = 4
225	for i in range(samples):		205 for i in range(samples):
226	x = 2 * (np.random.rand(dim)) - 1		206 x = 2 * (np.random.rand(dim)) - 1
227	y = 0		207 y = 0
228	for c, r in zip(centers, radii):		208 for c, r in zip(centers, radii):
229	if np.linalg.norm(x - c) < r:		209 if np.linalg.norm(x - c) < r:
230	y = 1		210 y = 1
231			211
232	data.append([x, y])		212 data.append([x, y])
233			213
234	return data, (centers, radii)		214 return data, (centers, radii)
235			215

1 2 3 4 5 6 7 8 9 10 11 12 13 14	<pre>#Quantum classifier #Sara Aminpour, Mike Banad, Sarah Sharif #September 25th 2024 #School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA, ##### #IMPORTANT_NOTE: #The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas. #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code. #Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem #as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods. from big_functions import minimizer, painter, SGD_step_by_step_minimization, overlearning_paint import datetime qubits = 2 #integer, number of qubits</pre>	<>		1 2 3	<pre>from big_functions import minimizer, painter, SGD_step_by_step_minimization, overlearning_paint, paint_world qubits = 1 #integer, number of qubits</pre>
15	<pre>layers = 5 #integer, number of layers (time we reupload data)</pre>	=		4	<pre>layers = 5 #integer, number of layers (time we reupload data)</pre>
16	<pre>chi = 'fidelity_chi' #Cost function; choose between ['fidelity_chi', 'trace_chi']</pre>	<>		5 6	<pre>chi = 'fidelity_chi' #Cost function; choose between ['fidelity_chi', 'weighted_fidelity_chi'] problem='wavy lines' #name of the problem, choose among ['circle', 'wavy circle', '3 circles', 'wavy lines', 'sphere', 'non convex', 'crown']</pre>
17	<pre>entanglement = 'y' #entanglement y/n</pre>	=		7	<pre>entanglement = 'y' #entanglement y/n</pre>
		-+		8	<pre>method = 'L-BFGS-B' #minimization methods, scipy methods or 'SGD'</pre>
18 19 20 21	<pre>name = 'run' #However you want to name your files seed = 30 #random seed #epochs=3000 #number of epochs, only for SGD methods</pre>	=		9 10 11 12	<pre>name = 'run' #However you want to name your files seed = 30 #random seed #epochs=3000 #number of epochs, only for SGD methods</pre>
22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>problem=['circle', 'line'] #name of the problem, choose among ['circle', 'wavy circle', '3 circles', 'wavy lines', 'sphere', 'non convex', 'crown'] for problem in problem: method = ['l-bfgs-b', 'cobyla', 'nelder-mead', 'slsqp'] #minimization methods between ['l-bfgs-b', 'cobyla', 'nelder-mead', 'slsqp'] for method in method: a=datetime.datetime.now() #SGD_step_by_step_minimization(problem, qubits, entanglement, layers, name) minimizer(chi, problem, qubits, entanglement, layers, method, name) painter(chi, problem, qubits, entanglement, layers, method, name, standard_test=True) #paint_world(chi, problem, qubits, entanglement, layers, method, name, standard_test=True) b=datetime.datetime.now() c=b-a text_file_nn = open('time.txt', mode='a+') text_file_nn.write(problem + '_' + chi + '_' + method + '_' + str(qubits) + 'Qubits_' + entanglement + '_' + str(layers) +'Layers_' + method + "_" + 'total_time'+ '=' + str(c)) text_file_nn.write('\n') text_file_nn.write('=====') text_file_nn.write('\n') text_file_nn.close()</pre>	<>		13 14 15 16	<pre>#SGD_step_by_step_minimization(problem, qubits, entanglement, layers, name) minimizer(chi, problem, qubits, entanglement, layers, method, name, seed = seed) painter(chi, problem, qubits, entanglement, layers, method, name, standard_test=True, seed=seed) paint_world(chi, problem, qubits, entanglement, layers, method, name, standard_test=True, seed=seed)</pre>

1	# coding=utf-8	+-		1	#####
2	#####	=		2	Quantum classifier
3	Quantum classifier			3	Adrián Pérez-Salinas, Alba Cervera-Liarta, Elies Gil, J. Ignacio Latorre
4	Sara Aminpour, Mike Banad, Sarah Sharif	<>		4	Code by APS
5	September 25th 2024			5	Code-checks by ACL
				6	June 3rd 2019
6		=		7	
7	#School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,	<>		8	
8	#####				
9	IMPORTANT NOTE:				
10	The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas.				
11	The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code				
12	Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem			9	Universitat de Barcelona / Barcelona Supercomputing Center/Institut de Ciències del Cosmos
13	as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.			10	
14	#####	=		11	#####
15	## This file creates the problems and their settings			12	
16	import numpy as np			13	## This file creates the problems and their settings
17				14	import numpy as np
18				15	
19	def problem_generator(problem, qubits, layers, chi, qubits_lab=1):			16	def problem_generator(problem, qubits, layers, chi, qubits_lab=1):
20	"""			17	"""
21	This function generates everything needed for solving the problem			18	This function generates everything needed for solving the problem
22	INPUT:			19	INPUT:
23	-chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'			20	-chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
24	-problem: name of the problem, to choose among			21	-problem: name of the problem, to choose among
25	['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines']			22	['circle', '3 circles', 'hypersphere', 'tricrown', 'non convex', 'crown', 'sphere', 'squares', 'wavy lines']
26	-qubits: number of qubits, must be an integer			23	-qubits: number of qubits, must be an integer
27	-layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account			24	-layers: number of layers, must be an integer. If layers == 1, entanglement is not taken in account
28				25	
29				26	
30	OUTPUT:			27	OUTPUT:
31	-theta: set of parameters needed for the circuit. It is an array with shape (qubits, layers, 3)			28	-theta: set of parameters needed for the circuit. It is an array with shape (qubits, layers, 3)
32	-alpha: set of parameters needed for the circuit. It is an array with shape (qubits, layers, dimension of data)			29	-alpha: set of parameters needed for the circuit. It is an array with shape (qubits, layers, dimension of data)
33	-weight: set of parameters needed fot the circuit only if chi == 'weighted_fidelity_chi'. It is an array with shape (classes, qubits)			30	-weight: set of parameters needed fot the circuit only if chi == 'weighted_fidelity_chi'. It is an array with shape (classes, qubits)
34	-reprs: variable encoding the label states of the different classes			31	-reprs: variable encoding the label states of the different classes
35	"""			32	"""
36	chi = chi.lower()			33	chi = chi.lower()
37	if chi in ['fidelity', 'weighted_fidelity', 'trace']: chi += ' chi'	<>		34	if chi in ['fidelity', 'weighted_fidelity']: chi += ' chi'
38	if chi not in ['fidelity_chi', 'weighted_fidelity_chi', 'trace_chi']:			35	if chi not in ['fidelity_chi', 'weighted_fidelity_chi']:
39	raise ValueError('Figure of merit is not valid')	=		36	raise ValueError('Figure of merit is not valid')
40				37	
41	if chi == 'weighted_fidelity_chi' and qubits_lab != 1:			38	if chi == 'weighted_fidelity_chi' and qubits_lab != 1:
42	qubits_lab = 1			39	qubits_lab = 1
43	print("WARNING: number of qubits for the label states has been changed to 1")			40	print("WARNING: number of qubits for the label states has been changed to 1")
44				41	
45	problem = problem.lower()			42	problem = problem.lower()
46	if problem == 'circle':			43	if problem == 'circle':
47	theta, alpha, reprs = _circle(qubits, layers, qubits_lab, chi)			44	theta, alpha, reprs = _circle(qubits, layers, qubits_lab, chi)
48	elif problem == '3 circles':			45	elif problem == '3 circles':
49	theta, alpha, reprs = _3_circles(qubits, layers, qubits_lab, chi)			46	theta, alpha, reprs = _3_circles(qubits, layers, qubits_lab, chi)
50	elif problem == 'wavy lines':			47	elif problem == 'wavy lines':
51	theta, alpha, reprs = _wavy_lines(qubits, layers, qubits_lab, chi)			48	theta, alpha, reprs = _wavy_lines(qubits, layers, qubits_lab, chi)
52	elif problem == 'squares':			49	elif problem == 'squares':
53	theta, alpha, reprs = _squares(qubits, layers, qubits_lab, chi)			50	theta, alpha, reprs = _squares(qubits, layers, qubits_lab, chi)
54	elif problem == 'sphere':			51	elif problem == 'sphere':
55	theta, alpha, reprs = _sphere(qubits, layers, qubits_lab, chi)			52	theta, alpha, reprs = _sphere(qubits, layers, qubits_lab, chi)
56	elif problem == 'non convex':			53	elif problem == 'non convex':
57	theta, alpha, reprs = _non_convex(qubits, layers, qubits_lab, chi)			54	theta, alpha, reprs = _non_convex(qubits, layers, qubits_lab, chi)
58	elif problem == 'crown':			55	elif problem == 'crown':
59	theta, alpha, reprs = _crown(qubits, layers, qubits_lab, chi)			56	theta, alpha, reprs = _crown(qubits, layers, qubits_lab, chi)
60	elif problem == 'tricrown':			57	elif problem == 'tricrown':
61	theta, alpha, reprs = _tricrown(qubits, layers, qubits_lab, chi)			58	theta, alpha, reprs = _tricrown(qubits, layers, qubits_lab, chi)
62	elif problem == 'hypersphere':			59	elif problem == 'hypersphere':
63	theta, alpha, reprs = hypersphere(qubits, layers, qubits_lab, chi)			60	theta, alpha, reprs = hypersphere(qubits, layers, qubits_lab, chi)
64	#####	<>		61	
65	elif problem == 'line':				
66	theta, alpha, reprs = _line(qubits, layers, qubits_lab, chi)				
67	#####				
68	else:	=		62	else:
69	raise ValueError('Problem is not valid')			63	raise ValueError('Problem is not valid')
70				64	
71	if chi == 'fidelity_chi':			65	if chi == 'fidelity_chi':
72	return theta, alpha, reprs			66	return theta, alpha, reprs
73	elif chi == 'trace_chi':	+-			
74	return theta, alpha, reprs				
75	elif chi == 'weighted_fidelity_chi':	=		67	elif chi == 'weighted_fidelity_chi':
76	weights = np.ones((len(reprs), qubits))			68	weights = np.ones((len(reprs), qubits))
77	return theta, alpha, weights, reprs			69	return theta, alpha, weights, reprs
78				70	
79	#All these are auxiliary functions for problem_generator			71	#All these are auxiliary functions for problem_generator
80	def _circle(qubits, layers, qubits_lab, chi):			72	def _circle(qubits, layers, qubits_lab, chi):
81	Classes = 2			73	Classes = 2
82	if chi == 'trace_chi':	<>			
83	reprs = representatives_tr(classes, qubits_lab)				
84	else:				
85	reprs = representatives(classes, qubits_lab)			74	reprs = representatives(classes, qubits_lab)
86					
87	theta = np.random.rand(qubits, layers, 3)	=		75	theta = np.random.rand(qubits, layers, 3)
88	alpha = np.random.rand(qubits, layers, 2)			76	alpha = np.random.rand(qubits, layers, 2)
89	return theta, alpha, reprs			77	return theta, alpha, reprs
90				78	
91	def _3_circles(qubits, layers, qubits_lab, chi):			79	def _3_circles(qubits, layers, qubits_lab, chi):
92	Classes = 4			80	Classes = 4
93	reprs = representatives(classes, qubits_lab)			81	reprs = representatives(classes, qubits_lab)
94	theta = np.random.rand(qubits, layers, 3)			82	theta = np.random.rand(qubits, layers, 3)
95	alpha = np.random.rand(qubits, layers, 2)			83	alpha = np.random.rand(qubits, layers, 2)
96	return theta, alpha, reprs			84	return theta, alpha, reprs
97				85	
98	def _wavy_lines(qubits, layers, qubits_lab, chi):			86	def _wavy_lines(qubits, layers, qubits_lab, chi):
99	Classes = 4			87	Classes = 4
100	reprs = representatives(classes, qubits_lab)			88	reprs = representatives(classes, qubits_lab)
101	theta = np.random.rand(qubits, layers, 3)			89	theta = np.random.rand(qubits, layers, 3)
102	alpha = np.random.rand(qubits, layers, 2)			90	alpha = np.random.rand(qubits, layers, 2)
103	return theta, alpha, reprs			91	return theta, alpha, reprs
104				92	
105	def _squares(qubits, layers, qubits_lab, chi):			93	def _squares(qubits, layers, qubits_lab, chi):
106	Classes = 4			94	Classes = 4
107	reprs = representatives(classes, qubits_lab)			95	reprs = representatives(classes, qubits_lab)
108	theta = np.random.rand(qubits, layers, 3)			96	theta = np.random.rand(qubits, layers, 3)
109	alpha = np.random.rand(qubits, layers, 2)			97	alpha = np.random.rand(qubits, layers, 2)
110	return theta, alpha, reprs			98	return theta, alpha, reprs
111	#####	<>			
112	def _line(qubits, layers, qubits_lab, chi):				
113	Classes = 2				
114	if chi == 'trace_chi':				
115	reprs = representatives_tr(classes, qubits_lab)				
116	else:				
117	reprs = representatives(classes, qubits_lab)				
118					
119	theta = np.random.rand(qubits, layers, 3)			99	
120	alpha = np.random.rand(qubits, layers, 2)				
121	return theta, alpha, reprs				
122	#####				
123	def _non_convex(qubits, layers, qubits_lab, chi):	=		100	def _non_convex(qubits, layers, qubits_lab, chi):
124	Classes = 2			101	Classes = 2
125	if chi == 'trace_chi':	<>			
126	reprs = representatives_tr(classes, qubits_lab)				
127	else:				
128	reprs = representatives(classes, qubits_lab)			102	reprs = representatives(classes, qubits_lab)
129					
130	theta = np.random.rand(qubits, layers, 3)	=		103	theta = np.random.rand(qubits, layers, 3)
131	alpha = np.random.rand(qubits, layers, 2)			104	alpha = np.random.rand(qubits, layers, 2)
132	return theta, alpha, reprs			105	return theta, alpha, reprs
133				106	
134	def _crown(qubits, layers, qubits_lab, chi):			107	def _crown(qubits, layers, qubits_lab, chi):
135	Classes = 2			108	Classes = 2
136	if chi == 'trace_chi':	<>			
137	reprs = representatives_tr(classes, qubits_lab)				
138	else:				
139	reprs = representatives(classes, qubits_lab)			109	reprs = representatives(classes, qubits_lab)
140					
141	theta = np.random.rand(qubits, layers, 3)	=		110	theta = np.random.rand(qubits, layers, 3)
142	alpha = np.random.rand(qubits, layers, 2)			111	alpha = np.random.rand(qubits, layers, 2)
143	return theta, alpha, reprs			112	return theta, alpha, reprs
144				113	
145	def _tricrown(qubits, layers, qubits_lab, chi):			114	def _tricrown(qubits, layers, qubits_lab, chi):
146	Classes = 3			115	Classes = 3
147	reprs = representatives(classes, qubits_lab)			116	reprs = representatives(classes, qubits_lab)
148	theta = np.random.rand(qubits, layers, 3)			117	theta = np.random.rand(qubits, layers, 3)
149	alpha = np.random.rand(qubits, layers, 2)			118	alpha = np.random.rand(qubits, layers, 2)
150	return theta, alpha, reprs			119	return theta, alpha, reprs
151				120	
152	def _sphere(qubits, layers, qubits_lab, chi):			121	def _sphere(qubits, layers, qubits_lab, chi):
153	Classes = 2			122	Classes = 2
154	reprs = representatives(classes, qubits_lab)			123	reprs = representatives(classes, qubits_lab)
155	theta = np.random.rand(qubits, layers, 3)			124	theta = np.random.rand(qubits, layers, 3)
156	alpha = np.random.rand(qubits, layers, 2)			125	alpha = np.random.rand(qubits, layers, 2)
157	return theta, alpha, reprs			126	return theta, alpha, reprs
158				127	
159	def _hypersphere(qubits, layers, qubits_lab, chi):			128	def _hypersphere(qubits, layers, qubits_lab, chi):
160	Classes = 2			129	Classes = 2
161	reprs = representatives(classes, qubits_lab)			130	reprs = representatives(classes, qubits_lab)
162	theta = np.random.rand(qubits, layers, 6)			131	theta = np.random.rand(qubits, layers, 6)
163	alpha = np.random.rand(qubits, layers, 4)			132	alpha = np.random.rand(qubits, layers, 4)
164	return theta, alpha, reprs			133	return theta, alpha, reprs
165				134	
166		+-			
167	def representatives_tr(classes, qubits_lab):				
168	"""				
169	This function creates the label states for the classification task				
170	INPUT:				
171	-classes: number of classes of our problem				
172	-qubits_lab: how many qubits will store the labels				
173	OUTPUT:				
174	-reprs: the label states				
175	"""				
176	#reprs = np.zeros((classes, 2**qubits_lab), dtype = 'complex')				
177	reprs = np.zeros((classes, 3), dtype = 'complex')				
178	if qubits_lab == 1:				
179	if classes == 0:				
180	raise ValueError('Nonsense classifier')				
181	if classes == 1:				
182	raise ValueError('Nonsense classifier')				
183	if classes == 2:				
184	#reprs[0] = np.array([1, 0])				
185	reprs[0] = np.array([0.293892621462367, -0.5090369604551273, 0.8090169943749473])				
186	reprs[1] = np.array([-0.293892621462367, 0.5090369604551273, -0.8090169943749473])				
187	if classes == 3:				
188	reprs[0] = np.array([1, 0])				
189	reprs[1] = np.array([1 / 2, np.sqrt(3) / 2])				
190	reprs[2] = np.array([1 / 2, -np.sqrt(3) / 2])				
191	if classes == 4:				
192	reprs[0] = np.array([1, 0])				
193	reprs[1] = np.array([1 / np.sqrt(3), np.sqrt(2 / 3)])				
194	reprs[2] = np.array([1 / np.sqrt(3), np.exp(1j * 2 * np.pi / 3) * np.sqrt(2 / 3)])				
195	reprs[3] = np.array([1 / np.sqrt(3), np.exp(-1j * 2 * np.pi / 3) * np.sqrt(2 / 3)])				
196	if classes == 6:				
197	reprs[0] = np.array([0.293892621462367, -0.5090369604551273, 0.8090169943749473])				
198	reprs[1] = np.array([-0.293892621462367, 0.5090369604551273, -0.8090169943749473])				
199	reprs[2] = np.array([-0.700629269220369, -0.4045084971874737, 0.5877852522924729])				
200	reprs[3] = np.array([0.700629269220369, 0.4045084971874737, -0.5877852522924729])				
201	reprs[4] = np.array([0.4045084971874736, -0.700629269220369, 0.5877852522924729])				
202	reprs[5] = np.array([0.700629269220369, 0.4045084971874737, -0.5877852522924729])				
203					
204	if qubits_lab == 2:				
205	if classes == 0:				
206	raise ValueError('Nonsense classifier')				
207	if classes == 1:				
208	raise ValueError('Nonsense classifier')				
209	if classes == 2:				
210	reprs[0] = np.array([0.29, -0.5, 0.8])				
211	reprs[1] = np.array([-0.29, 0.5, -0.8])				
212	if classes == 3:				
213	reprs[0] = np.array([1, 0, 0, 0])				
214	reprs[1] = np.array([0, 1, 0, 0])				
215	reprs[2] = np.array([0, 0, 1, 0])				
216	if classes == 4:				
217	reprs[0] = np.array([1, 0, 0, 0])				
218	reprs[1] = np.array([0, 1, 0, 0])				
219	reprs[2] = np.array([0, 0, 1, 0])				
220	reprs[3] = np.array([0, 0, 0, 1])				
221					
222	return reprs				
223					
224	def representatives(classes, qubits_lab):	=		135	def representatives(classes, qubits_lab):
225	"""			136	"""
226	This function creates the label states for the classification task			137	This function creates the label states for the classification task
227	INPUT:			138	INPUT:
228	-classes: number of classes of our problem			139	-classes: number of classes of our problem
229	-qubits_lab: how many qubits will store the labels			140	-qubits_lab: how many qubits will store the labels
230	OUTPUT:			141	OUTPUT:
231	-reprs: the label states			142	-reprs: the label states
232	"""			143	"""
233	reprs = np.zeros((classes, 2**qubits_lab), dtype = 'complex')			144	reprs = np.zeros((classes, 2**qubits_lab), dtype = 'complex')
234	if qubits_lab == 1:			145	if qubits_lab == 1:
235	if classes == 0:			146	if classes == 0:
236	raise ValueError('Nonsense classifier')			147	raise ValueError('Nonsense classifier')

1	# coding=utf-8	+-	
2	#####	=	1 #####
3	Quantum classifier		2 #Quantum classifier
4	#Sara Aminpour, Mike Banad, Sarah Sharif	<>	3 #Adrián Pérez-Salinas, Alba Cervera-Lierta, Elies Gil, J. Ignacio Latorre
5	#September 25th 2024		4 #Code by APS
			5 #Code-checks by ACL
6		=	6 #June 3rd 2019
7	#School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,	<>	8
8	#####		
9	IMPORTANT NOTE:		
10	#The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas.		
11	#The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.		
12	#Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem		9 #Universitat de Barcelona / Barcelona Supercomputing Center/Institut de Ciències del Cosmos
13	#as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLQP minimization methods.		10
14	#####	=	11 #####
15			12
16			13
17	## This is an auxiliary file. It provides the tools needed for simulating quantum		14 ## This is an auxiliary file. It provides the tools needed for simulating quantum
18	# circuits.		15 # circuits.
19			16
20	import numpy as np		17 import numpy as np
21	class QCircuit(object):		18 class QCircuit(object):
22	def __init__(self,qubits):		19 def __init__(self,qubits):
23	self.num_qubits = qubits		20 self.num_qubits = qubits
24	self.psi = [0]*2**self.num_qubits		21 self.psi = [0]*2**self.num_qubits
25	self.psi[0] = 1		22 self.psi[0] = 1
26	self.E_x=0		23 self.E_x=0
27	self.E_y=0		24 self.E_y=0
28	self.E_z=0		25 self.E_z=0
29	self.r=np.array([0,0,0])	+-	
30		=	26
31	def Ry(self,i,theta):		27 def Ry(self,i,theta):
32	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		28 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
33	c = np.cos(theta/2)		29 c = np.cos(theta/2)
34	s = np.sin(theta/2)		30 s = np.sin(theta/2)
35	for k in range(2**(self.num_qubits-1)):		31 for k in range(2**(self.num_qubits-1)):
36	S = k%(2**i) + 2*(k - k%(2**i))		32 S = k%(2**i) + 2*(k - k%(2**i))
37	S_ = S + 2**i		33 S_ = S + 2**i
38	a=c*self.psi[S] - s*self.psi[S_];		34 a=c*self.psi[S] - s*self.psi[S_];
39	b=s*self.psi[S] + c*self.psi[S_];		35 b=s*self.psi[S] + c*self.psi[S_];
40	self.psi[S]=a; self.psi[S_]=b;		36 self.psi[S]=a; self.psi[S_]=b;
41			37
42	def Rx(self,i,theta):		38 def Rx(self,i,theta):
43	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		39 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
44	c = np.cos(theta/2)		40 c = np.cos(theta/2)
45	s = np.sin(theta/2)		41 s = np.sin(theta/2)
46	for k in range(2**(self.num_qubits-1)):		42 for k in range(2**(self.num_qubits-1)):
47	S = k%(2**i) + 2*(k - k%(2**i))		43 S = k%(2**i) + 2*(k - k%(2**i))
48	S_ = S + 2**i		44 S_ = S + 2**i
49	a=c*self.psi[S] - lj*s*self.psi[S_];		45 a=c*self.psi[S] - lj*s*self.psi[S_];
50	b=-lj*s*self.psi[S] + c*self.psi[S_];		46 b=-lj*s*self.psi[S] + c*self.psi[S_];
51	self.psi[S]=a; self.psi[S_]=b;		47 self.psi[S]=a; self.psi[S_]=b;
52			48
53	def U2(self,i,phi,lamb):		49 def U2(self,i,phi,lamb):
54	if i >= self.num_qubits: raise ValueError('There are not enough qubits')		50 if i >= self.num_qubits: raise ValueError('There are not enough qubits')
55	f = np.exp(1j*phi)		51 f = np.exp(1j*phi)
56	l = np.exp(-1j*lamb)		52 l = np.exp(-1j*lamb)
57	for k in range(2**(self.num_qubits-1)):		53 for k in range(2**(self.num_qubits-1)):
58	S = k%(2**i) + 2*(k - k%(2**i))		54 S = k%(2**i) + 2*(k - k%(2**i))
59	S_ = S + 2**i		55 S_ = S + 2**i
60	a=1/np.sqrt(2)*(self.psi[S] - l*self.psi[S_]);		56 a=1/np.sqrt(2)*(self.psi[S] - l*self.psi[S_]);
61	b=1/np.sqrt(2)*(f*self.psi[S] + f*l*self.psi[S_]);		57 b=1/np.sqrt(2)*(f*self.psi[S] + f*l*self.psi[S_]);
62	self.psi[S]=a; self.psi[S_]=b;		58 self.psi[S]=a; self.psi[S_]=b;
63			59
64	def U3(self, i, theta3):		60 def U3(self, i, theta3):
65	if i >= self.num_qubits: raise ValueError('There are not enough qubits')		61 if i >= self.num_qubits: raise ValueError('There are not enough qubits')
66	c = np.cos(theta3[0] / 2)		62 c = np.cos(theta3[0] / 2)
67	s = np.sin(theta3[0] / 2)		63 s = np.sin(theta3[0] / 2)
68	e_phi = np.exp(1j * theta3[1] / 2)		64 e_phi = np.exp(1j * theta3[1] / 2)
69	e_phi_s = np.conj(e_phi)		65 e_phi_s = np.conj(e_phi)
70	e_lambda = np.exp(1j * theta3[2] / 2)		66 e_lambda = np.exp(1j * theta3[2] / 2)
71	e_lambda_s = np.conj(e_lambda)		67 e_lambda_s = np.conj(e_lambda)
72		+-	
73	for k in range(2 ** (self.num_qubits - 1)):	=	68 for k in range(2 ** (self.num_qubits - 1)):
74	S = k % (2 ** i) + 2 * (k - k % (2 ** i))		69 S = k % (2 ** i) + 2 * (k - k % (2 ** i))
75	S_ = S + 2 ** i		70 S_ = S + 2 ** i
76	a = c * e_phi * e_lambda * self.psi[S] - s * e_phi * e_lambda_s * self.psi[S_];		71 a = c * e_phi * e_lambda * self.psi[S] - s * e_phi * e_lambda_s * self.psi[S_];
77	b = s * e_phi_s * e_lambda * self.psi[S] + c * e_phi_s * e_lambda_s * self.psi[S_];		72 b = s * e_phi_s * e_lambda * self.psi[S] + c * e_phi_s * e_lambda_s * self.psi[S_];
78	self.psi[S] = a;		73 self.psi[S] = a;
79	self.psi[S_] = b;		74 self.psi[S_] = b;
80			75
81	theta_f=np.arccos(np.abs(self.psi[S])**2 - np.abs(self.psi[S_])**2) - np.pi/2	+-	
82	phi_f=np.angle(self.psi[S_] / self.psi[S])		
83	self.r=np.array((np.sin(theta_f)*np.cos(phi_f),np.sin(phi_f)*np.sin(theta_f),np.cos(theta_f)))		
84			
85	def Rz(self,i,theta):	=	76 def Rz(self,i,theta):
86	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		77 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
87	ex = np.exp(1j*theta)		78 ex = np.exp(1j*theta)
88	for k in range(2**(self.num_qubits-1)):		79 for k in range(2**(self.num_qubits-1)):
89	S = k%(2**i) + 2*(k - k%(2**i)) + 2**i		80 S = k%(2**i) + 2*(k - k%(2**i)) + 2**i
90	self.psi[S]=ex*self.psi[S];		81 self.psi[S]=ex*self.psi[S];
91			82
92	def Hx(self,i):		83 def Hx(self,i):
93	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		84 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
94	for k in range(2**(self.num_qubits-1)):		85 for k in range(2**(self.num_qubits-1)):
95	S = k%(2**i) + 2*(k - k%(2**i))		86 S = k%(2**i) + 2*(k - k%(2**i))
96	S_ = S + 2**i		87 S_ = S + 2**i
97	a=1/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];		88 a=1/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
98	b=1/np.sqrt(2)*self.psi[S] - 1/np.sqrt(2)*self.psi[S_];		89 b=1/np.sqrt(2)*self.psi[S] - 1/np.sqrt(2)*self.psi[S_];
99	self.psi[S] = a		90 self.psi[S] = a
100	self.psi[S_] = b		91 self.psi[S_] = b
101			92
102	def Hy(self,i):		93 def Hy(self,i):
103	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		94 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
104	for k in range(2**(self.num_qubits-1)):		95 for k in range(2**(self.num_qubits-1)):
105	S = k%(2**i) + 2*(k - k%(2**i))		96 S = k%(2**i) + 2*(k - k%(2**i))
106	S_ = S + 2**i		97 S_ = S + 2**i
107	a=1/np.sqrt(2)*self.psi[S] -1j/np.sqrt(2)*self.psi[S_];		98 a=1/np.sqrt(2)*self.psi[S] -1j/np.sqrt(2)*self.psi[S_];
108	b = 1j/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];		99 b = 1j/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
109	self.psi[S] = a		100 self.psi[S] = a
110	self.psi[S_] = b		101 self.psi[S_] = b
111			102
112	def HyT(self,i):		103 def HyT(self,i):
113	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		104 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
114	for k in range(2**(self.num_qubits-1)):		105 for k in range(2**(self.num_qubits-1)):
115	S = k%(2**i) + 2*(k - k%(2**i))		106 S = k%(2**i) + 2*(k - k%(2**i))
116	S_ = S + 2**i		107 S_ = S + 2**i
117	a=1/np.sqrt(2)*self.psi[S] +1j/np.sqrt(2)*self.psi[S_];		108 a=1/np.sqrt(2)*self.psi[S] +1j/np.sqrt(2)*self.psi[S_];
118	b=1j/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];		109 b=1j/np.sqrt(2)*self.psi[S] + 1/np.sqrt(2)*self.psi[S_];
119	self.psi[S]=a; self.psi[S_]=b;		110 self.psi[S]=a; self.psi[S_]=b;
120			111
121	def Cz(self,i,j):		112 def Cz(self,i,j):
122	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		113 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
123	if j>=self.num_qubits: raise ValueError('There are not enough qubits')		114 if j>=self.num_qubits: raise ValueError('There are not enough qubits')
124	if i==j: raise ValueError('Control and target qubits are the same')		115 if i==j: raise ValueError('Control and target qubits are the same')
125	if j<i: a=1; i=j; j=a;		116 if j<i: a=1; i=j; j=a;
126	for k in range(2**(self.num_qubits-2)):		117 for k in range(2**(self.num_qubits-2)):
127	S = k%2**i + (118 S = k%2**i + (
128	(k - k%2**i)*2)%2**j + 2*(119 (k - k%2**i)*2)%2**j + 2*(
129	(k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i + 2**j;		120 (k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i + 2**j;
130	self.psi[S]=self.psi[S]		121 self.psi[S]=self.psi[S]
131			122
132	def SWAP(self,i,j):		123 def SWAP(self,i,j):
133	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		124 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
134	if j>=self.num_qubits: raise ValueError('There are not enough qubits')		125 if j>=self.num_qubits: raise ValueError('There are not enough qubits')
135	if i==j: raise ValueError('Control and target qubits are the same')		126 if i==j: raise ValueError('Control and target qubits are the same')
136	for k in range(2**(self.num_qubits-2)):		127 for k in range(2**(self.num_qubits-2)):
137	S = k%2**i + (128 S = k%2**i + (
138	(k - k%2**i)*2)%2**j + 2*(129 (k - k%2**i)*2)%2**j + 2*(
139	(k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**j;		130 (k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**j;
140	S_ = S + 2**i - 2**j		131 S_ = S + 2**i - 2**j
141	a=self.psi[S_]		132 a=self.psi[S_]
142	self.psi[S_] = self.psi[S]		133 self.psi[S_] = self.psi[S]
143	self.psi[S] = a		134 self.psi[S] = a
144			135
145			136
146	def Cx(self,i,j):		137 def Cx(self,i,j):
147	#i = control		138 #i = control
148	#j = target		139 #j = target
149	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		140 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
150	if j>=self.num_qubits: raise ValueError('There are not enough qubits')		141 if j>=self.num_qubits: raise ValueError('There are not enough qubits')
151	if i==j: raise ValueError('Control and target qubits are the same')		142 if i==j: raise ValueError('Control and target qubits are the same')
152	for k in range(2**(self.num_qubits-2)):		143 for k in range(2**(self.num_qubits-2)):
153	S = k%2**i + (144 S = k%2**i + (
154	(k - k%2**i)*2)%2**j + 2*(145 (k - k%2**i)*2)%2**j + 2*(
155	(k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i;		146 (k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i;
156	S_ = S + 2**j		147 S_ = S + 2**j
157	.,.,.		148 .,.,.
158	a=self.psi[S_]		149 a=self.psi[S_]
159	self.psi[S_] = self.psi[S]		150 self.psi[S_] = self.psi[S]
160	self.psi[S] = a		151 self.psi[S] = a
161	.,.,.		152 .,.,.
162	self.psi[S],self.psi[S_] = self.psi[S_],self.psi[S]		153 self.psi[S],self.psi[S_] = self.psi[S_],self.psi[S]
163	def Cy(self,i,j):		154 def Cy(self,i,j):
164	if i>=self.num_qubits: raise ValueError('There are not enough qubits')		155 if i>=self.num_qubits: raise ValueError('There are not enough qubits')
165	if j>=self.num_qubits: raise ValueError('There are not enough qubits')		156 if j>=self.num_qubits: raise ValueError('There are not enough qubits')
166	if i==j: raise ValueError('Control and target qubits are the same')		157 if i==j: raise ValueError('Control and target qubits are the same')
167	for k in range(2**(self.num_qubits-2)):		158 for k in range(2**(self.num_qubits-2)):
168	S = k%2**i + (159 S = k%2**i + (
169	(k - k%2**i)*2)%2**j + 2*(160 (k - k%2**i)*2)%2**j + 2*(
170	(k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i;		161 (k-k%2**i)*2-((2*(k-k%2**i))%2**j)) + 2**i;
171	S_ = S + 2**j		162 S_ = S + 2**j
172	self.psi[S],self.psi[S_] = lj*self.psi[S_],-lj*self.psi[S]		163 self.psi[S],self.psi[S_] = lj*self.psi[S_],-lj*self.psi[S]
173			164
174	def MeasureZ(self):		165 def MeasureZ(self):
175	self.E_z = 0;		166 self.E_z = 0;
176	for h in range(2 ** self.num_qubits):		167 for h in range(2 ** self.num_qubits):
177	s = np.binary_repr(h, width=self.num_qubits)		168 s = np.binary_repr(h, width=self.num_qubits)
178	self.E_z += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))		169 self.E_z += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
179			170
180	def MeasureX(self):		171 def MeasureX(self):
181	self.E_x = 0;		172 self.E_x = 0;
182	for i in range(self.num_qubits):		173 for i in range(self.num_qubits):
183	self.Hx(i);		174 self.Hx(i);
184	for h in range(2 ** self.num_qubits):		175 for h in range(2 ** self.num_qubits):
185	s = np.binary_repr(h, width=self.num_qubits)		176 s = np.binary_repr(h, width=self.num_qubits)
186	self.E_x += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))		177 self.E_x += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
187	for i in range(self.num_qubits):		178 for i in range(self.num_qubits):
188	self.Hx(i);		179 self.Hx(i);
189			180
190	def MeasureY(self):		181 def MeasureY(self):
191	self.E_y = 0;		182 self.E_y = 0;
192	for i in range(self.num_qubits):		183 for i in range(self.num_qubits):
193	self.Hy(i);		184 self.Hy(i);
194	for h in range(2 ** self.num_qubits):		185 for h in range(2 ** self.num_qubits):
195	s = np.binary_repr(h, width=self.num_qubits)		186 s = np.binary_repr(h, width=self.num_qubits)
196	self.E_y += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))		187 self.E_y += np.abs(self.psi[h])**2*(s.count('1')-s.count('0'))
197	for i in range(self.num_qubits):		188 for i in range(self.num_qubits):
198	self.HyT(i);		189 self.HyT(i);
199			190
200	def reduced_density_matrix(self, q):		191 def reduced_density_matrix(self, q):
201	rho = np.zeros((2,2), dtype='complex')		192 rho = np.zeros((2,2), dtype='complex')
202	for i in range(2):		193 for i in range(2):
203	for j in range(i + 1):		194 for j in range(i + 1):
204	for k in range(2**(self.num_qubits-1)):		195 for k in range(2**(self.num_qubits-1)):
205	S = k%(2**q) + 2*(k - k%(2**q))		196 S = k%(2**q) + 2*(k - k%(2**q))
206	rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])		197 rho[i,j] += self.psi[S + i*2**q] * np.conj(self.psi[S + j*2**q])
207	rho[j,i] = np.conj(rho[i,j])		198 rho[j,i] = np.conj(rho[i,j])
208	return rho		199 return rho
209			200
210			201

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	149
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	-----

1	# coding=utf-8	+-	
2	#####	=	1 #####
3	#Quantum classifier		2 #Quantum classifier
4	#Sara Aminpour, Mike Banad, Sarah Sharif	<>	3 #Adrián Pérez-Salinas, Alba Cervera-Lierta, Elies Gil, J. Ignacio Latorre
5	#September 25th 2024		4 #Code by APS
			5 #Code-checks by ACL
6		=	6 #June 3rd 2019
7	#School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,	<>	7
8	#####		
9	#IMPORTANT NOTE:		
10	#The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas.		
11	#The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.		
12	#Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem		9 #Universitat de Barcelona / Barcelona Supercomputing Center/Institut de Ciències del Cosmos
13	as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.		10
14	#####	=	11 #####
15			
16			
17	#This file provides useful tools checking how good our results are		13
18			14 #This file provides useful tools checking how good our results are
19	from circuitery import code_coords, circuit		15
20	from fidelity minimization import fidelity		16 from circuitery import code_coords, circuit
21	from trace minimization import trace_dis	+-	17 from fidelity minimization import fidelity
22	from weighted_fidelity_minimization import mat_fidelities, w_fidelities		18 from weighted_fidelity_minimization import mat_fidelities, w_fidelities
23	import numpy as np		19 import numpy as np
24			20
25	def _claim(theta, alpha, weight, x, reprs, entanglement, chi):		21 def _claim(theta, alpha, weight, x, reprs, entanglement, chi):
26	"""		22 """
27	This function takes the parameters of a solved problem and one data computes classification of this point		23 This function takes the parameters of a solved problem and one data computes classification of this point
28	INPUT:		24 INPUT:
29	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		25 -theta: initial point for the theta parameters. The shape must be correct (qubits, layers,
30	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		26 3)
31	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		27 dim)
32	-x: coordinates of data for testing.		28 qubits)
33	-reprs: variable encoding the label states of the different classes		29 -x: coordinates of data for testing.
34	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		30 -reprs: variable encoding the label states of the different classes
35	-chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'		31 -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
36	OUTPUT:		32 -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
37	"""		33 OUTPUT:
38	-y_: the class of x, according to the classifier		34 """
39	chi = chi.lower().replace(' ', '')		35 -y_: the class of x, according to the classifier
40	if chi in ['fidelity', 'weighted_fidelity', 'trace']: chi += '_chi'	<>	36 if chi in ['fidelity', 'weighted_fidelity']: chi += '_chi'
41	if chi not in ['fidelity_chi', 'weighted_fidelity_chi', 'trace_chi']:		37 if chi not in ['fidelity_chi', 'weighted_fidelity_chi']:
42	raise ValueError('Figure of merit is not valid')	=	38 raise ValueError('Figure of merit is not valid')
43			39
44	if chi == 'fidelity_chi':		40 if chi == 'fidelity_chi':
45	y_ = _claim_fidelity(theta, alpha, x, reprs, entanglement)		41 y_ = _claim_fidelity(theta, alpha, x, reprs, entanglement)
46			42
47	if chi == 'trace_chi':	+-	
48	y_ = _claim_trace(theta, alpha, x, reprs, entanglement)		
49			
50	if chi == 'weighted_fidelity_chi':	=	43 if chi == 'weighted_fidelity_chi':
51	y_ = _claim_weighted_fidelity(theta, alpha, weight, x, reprs, entanglement)		44 y_ = _claim_weighted_fidelity(theta, alpha, weight, x, reprs, entanglement)
52		<>	45
53	return y_	=	46 return y_
54			47
55			48
56	def _claim_fidelity(theta, alpha, x, reprs, entanglement):		49 def _claim_fidelity(theta, alpha, x, reprs, entanglement):
57	"""		50 """
58	This function is inside _claim for fidelity_chi		51 This function is inside _claim for fidelity_chi
59	INPUT:		52 INPUT:
60	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		53 -theta: initial point for the theta parameters. The shape must be correct (qubits, layers,
61	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		54 3)
62	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		55 dim)
63	-x: coordinates of data for testing.		56 qubits)
64	-reprs: variable encoding the label states of the different classes		57 -x: coordinates of data for testing.
65	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		58 -reprs: variable encoding the label states of the different classes
66	OUTPUT:		59 -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
67	the class of x, according to the classifier		60 OUTPUT:
68	"""		61 the class of x, according to the classifier
69	theta_aux = code_coords(theta, alpha, x)		62 """
70	C = circuit(theta_aux, entanglement)		63 theta_aux = code_coords(theta, alpha, x)
71	Fidelities = [fidelity(r, C.psi) for r in reprs]		64 C = circuit(theta_aux, entanglement)
		+-	65 Fidelities = [fidelity(r, C.psi) for r in reprs]
72	return np.argmax(Fidelities)	=	66 return np.argmax(Fidelities)
73		+-	
74			
75			
76			
77	#####		
78	#####		
79	#####		
80	#####		
81	#####		
82	#####		
83	def _claim_trace(theta, alpha, x, reprs, entanglement):		
84	"""		
85	This function is inside _claim for fidelity_chi		
86	INPUT:		
87	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		
88	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		
89	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		
90	-x: coordinates of data for testing.		
91	-reprs: variable encoding the label states of the different classes		
92	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		
93	OUTPUT:		
94	the class of x, according to the classifier		
95	"""		
96	theta_aux = code_coords(theta, alpha, x)		
97	C = circuit(theta_aux, entanglement)		
98	#for r1 in reprs:		
99	# Trace=trace_dis(r1, C.r)		
100	Trace = [trace_dis(r1, C.r) for r1 in reprs]		
101	#print('td=',Trace)		
102	#print('reprs[y]=' ,r1)		
103	#print('C.r=',C.r)		
104	#print('min=',np.argmin(Trace))		
105	return np.argmax(Trace)		
106			
107			
108	#####		
109	#####		
110	#####		
111	#####		
112	#####		
113	#####		
114			
115		=	67
116			68
117	def _claim_weighted_fidelity(theta, alpha, weight, x, reprs, entanglement):		69 def _claim_weighted_fidelity(theta, alpha, weight, x, reprs, entanglement):
118	"""		70 """
119	This function is inside _claim for weighted_fidelity_chi		71 This function is inside _claim for weighted_fidelity_chi
120	INPUT:		72 INPUT:
121	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		73 -theta: initial point for the theta parameters. The shape must be correct (qubits, layers,
122	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		74 3)
123	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		75 dim)
124	-x: coordinates of data for testing.		76 qubits)
125	-reprs: variable encoding the label states of the different classes		77 -x: coordinates of data for testing.
126	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		78 -reprs: variable encoding the label states of the different classes
127	OUTPUT:		79 -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
128	the class of x, according to the classifier		80 OUTPUT:
129	"""		81 the class of x, according to the classifier
130	theta_aux = code_coords(theta, alpha, x)		82 """
131	fids = mat_fidelities(theta_aux, weight, reprs, entanglement)		83 theta_aux = code_coords(theta, alpha, x)
132	w_fid = w_fidelities(fids, weight)		84 fids = mat_fidelities(theta_aux, weight, reprs, entanglement)
133	return np.argmax(w_fid)		85 w_fid = w_fidelities(fids, weight)
134			86 return np.argmax(w_fid)
135	def tester(theta, alpha, test_data, reprs, entanglement, chi, weights=None):		87 def tester(theta, alpha, test_data, reprs, entanglement, chi, weights=None):
136	"""		88 """
137	This function takes the parameters of a solved problem and one data computes how many points are correct		89 This function takes the parameters of a solved problem and one data computes how many points are correct
138	INPUT:		90 INPUT:
139	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		91 -theta: initial point for the theta parameters. The shape must be correct (qubits, layers,
140	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		92 3)
141	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		93 dim)
142	-test_data: set of data for testing		94 qubits)
143	-reprs: variable encoding the label states of the different classes		95 -test_data: set of data for testing
144	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		96 -reprs: variable encoding the label states of the different classes
145	-chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'		97 -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
146	OUTPUT:		98 -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
147	-success normalized		99 OUTPUT:
148	"""		100 -success normalized
149	acc = 0		101 """
150	for i, d in enumerate(test_data):		102 acc = 0
151	x, y = d		103 for i, d in enumerate(test_data):
152	y_ = _claim(theta, alpha, weights, x, reprs, entanglement, chi)		104 x, y = d
153	if y_ == y:		105 y_ = _claim(theta, alpha, weights, x, reprs, entanglement, chi)
154	acc += 1		106 if y_ == y:
		+-	107 acc += 1
155	return acc / len(test_data)	=	108 return acc / len(test_data)
156			109
157			110
158	def Accuracy_test(theta, alpha, test_data, reprs, entanglement, chi, weights=None):		111 def Accuracy_test(theta, alpha, test_data, reprs, entanglement, chi, weights=None):
159	"""		112 """
160	This function takes the parameters of a solved problem and one data computes how many points are correct		113 This function takes the parameters of a solved problem and one data computes how many points are correct
161	INPUT:		114 INPUT:
162	-theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)		115 -theta: initial point for the theta parameters. The shape must be correct (qubits, layers,
163	-alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)		116 3)
164	-weight: set of parameters needed fot the circuit. Must be an array with shape (classes, qubits)		117 dim)
165	-test_data: set of data for testing		118 qubits)
166	-reprs: variable encoding the label states of the different classes		119 -test_data: set of data for testing
167	-entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'		120 -reprs: variable encoding the label states of the different classes
168	-chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'		121 -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
169	OUTPUT:		122 -chi: cost function, to choose between 'fidelity_chi' or 'weighted_fidelity_chi'
170	-solutions of the classification		123 OUTPUT:
171	-success normalized		124 -solutions of the classification
172	"""		125 -success normalized
173	dim = len(test_data[0][0])		126 """
174	solutions = np.zeros((len(test_data), dim + 3)) #data #Esto se podrá mejorar en el futuro		127 dim = len(test_data[0][0])
175	for i, d in enumerate(test_data):		128 solutions = np.zeros((len(test_data), dim + 3)) #data #Esto se podrá mejorar en el futuro
176	x, y = d		129 for i, d in enumerate(test_data):
177	y_ = _claim(theta, alpha, weights, x, reprs, entanglement, chi)		130 x, y = d
178	solutions[i,:dim] = x		131 y_ = _claim(theta, alpha, weights, x, reprs, entanglement, chi)
179	solutions[i, -3] = y		132 solutions[i,:dim] = x
180	solutions[i, -2] = y_		133 solutions[i, -3] = y
181	solutions[i, -1] = int(y == y_)		134 solutions[i, -2] = y_
182			135 solutions[i, -1] = int(y == y_)
183	acc = np.sum(solutions[:, -1]) / (i + 1)		136
184			137
185	return solutions, acc		138 acc = np.sum(solutions[:, -1]) / (i + 1)
186			139 return solutions, acc


```
1 # coding=utf-8
2 #####
3 #Quantum classifier
4 #Sara Aminpour, Mike Banad, Sarah Sharif
5 #September 25th 2024
6
7 #School of Electrical and Computer Engineering/ Center for Quantum and Technology, University of Oklahoma, Norman, OK 73019 USA,
8 #####
9 #IMPORTANT NOTE:
10 #The code on the left was developed by Sara Aminpour, while the code on the right serves as the reference implementation by Adrián Pérez-Salinas.
11 #The code on the left has been restructured to handle random data. So some certain sections has been deleted from the reference code.
12 Additionally, our code on the left developed to analyze trace distance cost function and linear classification problem
13 as well as necessary modification to apply COBYLA, L-BFGS-B, NELDER-MEAD, and SLSQP minimization methods.
14 #####
15
16
17 #This file provides the minimization for the cheap chi square
18 from circuitry import code_coords, circuit
19 import numpy as np
20 import random
21 from scipy.optimize import minimize
22
23 def trace_minimization(theta, alpha, train_data, reprs,
24                        entanglement, method,
25                        batch_size, eta, epochs):
26     """
27     This function takes the parameters of a problem and computes the optimal parameters for it, using different functions. It uses the trace minimization
28     INPUT:
29     -theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)
30     -alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)
31     -train_data: set of data for training. There must be several entries (x,y)
32     -reprs: variable encoding the label states of the different classes
33     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
34     -method: minimization method, to choose among ['SGD', another valid for function scipy.optimize.minimize]
35     -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method
36     -eta: learning rate, only for 'SGD' method
37     -epochs: number of epochs , only for 'SGD' method
38     OUTPUT:
39     -theta: optimized point for the theta parameters. The shape is correct (qubits, layers, 3)
40     -alpha: optimized point for the alpha parameters. The shape is correct (qubits, layers, dim)
41     -chi: value of the minimization function
42     """
43
44     if method == 'SGD':
45         thetas, alphas, chis = _sgd(theta, alpha, train_data, reprs,
46                                     entanglement, eta, batch_size, epochs)
47         i = chis.index(max(chis))
48         return thetas[i], alphas[i], chis[i]
49     else:
50         params, hypars = _translate_to_scipy(theta, alpha)
51         results = minimize(_scipy_minimizing, params,
52                           args = (hypars, train_data, reprs, entanglement),
53                           method=method)
54         theta, alpha = _translate_from_scipy(results['x'], hypars)
55
56     return theta, alpha, results['fun']
57
58
59 def _gradient(theta, alpha, data, reprs, entanglement):
60     """
61     This function computes a gradient step for the SGD minimization
62     INPUT:
63     -theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)
64     -alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)
65     -data: one data for training. It must be (x,y)
66     -reprs: variable encoding the label states of the different classes
67     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
68     OUTPUT:
69     -grad_theta: gradient for the theta parameters. The shape is correct (qubits, layers, 3)
70     -grad_alpha: gradient for the alpha parameters. The shape is correct (qubits, layers, dim)
71     -results['fun']: value of the minimization function
72     """
73
74     x,y = data
75     theta_aux = code_coords(theta, alpha, x)
76     C = circuit(theta_aux, entanglement)
77     prodl = np.dot(np.conj(reprs[y]), C.psi)
78     prods2 = np.zeros(theta.shape, dtype='complex')
79     (Q, L, I) = theta_aux.shape
80
81     for q in range(0):
82         for l in range(L):
83             for i in range(I):
84                 theta_aux[q, l, i] += np.pi
85                 der_c = circuit(theta_aux, entanglement)
86                 prodl2[q, l, i] = np.dot(reprs[y], np.conj(der_c.psi))
87                 theta_aux[q, l, i] -= np.pi
88             grad_theta = np.asfarray(np.real(prodl * prods2))
89             if len(x) <= 3:
90                 dim = len(x)
91                 grad_alpha = np.empty((theta.shape[0], theta.shape[1], dim))
92                 for q in range(0):
93                     for l in range(L):
94                         for i in range(dim):
95                             grad_alpha[q, l, i] = x[i] * grad_theta[q, l, i]
96
97     if len(x) == 4:
98         grad_alpha = np.empty((theta.shape[0], theta.shape[1], 4))
99         for q in range(0):
100             grad_alpha[q, l, 0] = x[0] * grad_theta[q, l, 0]
101             grad_alpha[q, l, 1] = x[1] * grad_theta[q, l, 1]
102             grad_alpha[q, l, 2] = x[2] * grad_theta[q, l, 3]
103             grad_alpha[q, l, 3] = x[3] * grad_theta[q, l, 4]
104
105     return grad_theta, grad_alpha
106
107
108 def _train_batch(theta, alpha, batch, reprs, entanglement):
109     """
110     This function computes a gradient step for a complete batch for the SGD minimization
111     INPUT:
112     -theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)
113     -alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)
114     -batch: small set of data for training. It must be several (x,y)
115     -reprs: variable encoding the label states of the different classes
116     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
117     OUTPUT:
118     -grad_theta: gradient for the theta parameters averaged in batch. The shape is correct (qubits, layers, 3)
119     -grad_alpha: gradient for the alpha parameters averaged in batch. The shape is correct (qubits, layers, dim)
120     """
121
122     gradient_theta = np.zeros(theta.shape)
123     gradient_alpha = np.zeros(alpha.shape)
124     for d in batch:
125         g_t, g_a = _gradient(theta, alpha, d, reprs, entanglement)
126         gradient_theta += g_t
127         gradient_alpha += g_a
128
129     return gradient_theta / len(batch), gradient_alpha / len(batch)
130
131
132 def _session_sgd(theta, alpha, train_data, reprs, entanglement, eta, batch_size):
133     """
134     This function computes a gradient descent step for all batches
135     INPUT:
136     -theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)
137     -alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)
138     -train_data: set of data for training. There must be several entries (x,y)
139     -reprs: variable encoding the label states of the different classes
140     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
141     -eta: learning rate, only for 'SGD' method
142     -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method
143     OUTPUT:
144     -theta: updated point for the theta parameters. The shape is correct (qubits, layers, 3)
145     -alpha: updated point for the alpha parameters. The shape is correct (qubits, layers, dim)
146     -Av_chi_square: value of the minimization function
147     """
148
149     batches = [train_data[k:k + batch_size] for k in range(0,
150                                     len(train_data), batch_size)]
151     for batch in batches:
152         gradient_theta_batch, gradient_alpha_batch = _train_batch(
153             theta, alpha, batch, reprs, entanglement)
154         theta += eta * gradient_theta_batch #This sign is very important, it is the difference between maximizing or minimizing.
155         alpha += eta * gradient_alpha_batch
156
157     return theta, alpha, Av_Tr(theta, alpha, train_data, reprs, entanglement)
158
159
160 def _sgd(theta, alpha, train_data, reprs, entanglement, eta, batch_size, epochs):
161     """
162     This function completes the whole SGD strategy
163     INPUT:
164     -theta: initial point for the theta parameters. The shape must be correct (qubits, layers, 3)
165     -alpha: initial point for the alpha parameters. The shape must be correct (qubits, layers, dim)
166     -train_data: set of data for training. There must be several entries (x,y)
167     -reprs: variable encoding the label states of the different classes
168     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
169     -method: minimization method, to choose among ['SGD', another valid for function scipy.optimize.minimize]
170     -batch_size: size of the batches for stochastic gradient descent, only for 'SGD' method
171     -eta: learning rate, only for 'SGD' method
172     -epochs: number of epochs , only for 'SGD' method
173     OUTPUT:
174     -thetas: optimized points for the theta parameters for all epochs. The shape is correct (qubits, layers, 3)
175     -alphas: optimized points for the alpha parameters for all epochs. The shape is correct (qubits, layers, dim)
176     -chis: value of the minimization function at every step
177     """
178
179     thetas = [np.empty(theta.shape)] * epochs
180     alphas = [np.empty(alpha.shape)] * epochs
181     chis = [0] * epochs
182     for e in range(epochs):
183         theta, alpha, chi = _session_sgd(theta, alpha, train_data, reprs,
184                                         entanglement, eta, batch_size)
185         thetas[e] = theta
186         alphas[e] = alpha
187         chis[e] = chi
188
189     theta = theta
190     alpha = alpha
191
192     #Storage for solution
193
194     #Next step initialization
195
196     return thetas, alphas, chis
197
198
199 def _translate_to_scipy(theta, alpha):
200     """
201     This function is a intermediate step for translating theta and alpha to a single variable for scipy.optimize.minimize
202     """
203     qubits = theta.shape[0]
204     layers = theta.shape[1]
205     dim = alpha.shape[-1]
206
207     return np.concatenate((theta.flatten(), alpha.flatten())), (qubits, layers, dim)
208
209
210 def _translate_from_scipy(params, hypars):
211     """
212     This function is a intermediate step for getting theta and alpha from a single variable for scipy.optimize.minimize
213     """
214     (qubits, layers, dim) = hypars
215     if dim <= 3:
216         theta = params[qubits * layers * 3:].reshape(qubits, layers, 3)
217         alpha = params[qubits * layers * 3: qubits * layers * 3 + qubits * layers * dim].reshape(qubits, layers, dim)
218     if dim == 4:
219         theta = params[qubits * layers * 6:].reshape(qubits, layers, 6)
220         alpha = params[qubits * layers * 6: qubits * layers * 6 + qubits * layers * dim].reshape(qubits, layers, dim)
221     return theta, alpha
222
223
224 #####
225
226 #Sara
227 #####
228 def _scipy_minimizing(params, hypars, train_data, reprs, entanglement):
229     """
230     This function returns the chi^2 function for using scipy
231     INPUT:
232     -params: theta and alpha inside the same variable
233     -hypars: hyperparameters needed to rebuild theta and alpha
234     -train_data: training dataset for the classifier
235     -reprs: variable encoding the label states of the different classes
236     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
237     OUTPUT:
238     - - Av_Tr, which is the function we want to minimize
239     """
240     theta, alpha = _translate_from_scipy(params, hypars)
241     return -Av_Tr(theta, alpha, train_data, reprs, entanglement)
242
243
244 #Sara
245 #####
246
247 def trace_dis(r,s):
248     """
249     This function returns the trace distance of two pure states
250     INPUT:
251     -r,s: 2 vectors of pure states of the same dimension
252     OUTPUT:
253     -trace distance
254     """
255     dist = np.linalg.norm(r - s)
256     td=dist/2
257
258     return td
259
260
261
262 def _Tr(theta, alpha, data, reprs, entanglement): #Chi for one point
263     """
264     This function compute chi^2 for only one point
265     INPUT:
266     -theta: set of parameters needed for the circuit. Must be an array with shape (qubits, layers, 3)
267     -alpha: set of parameters needed for the circuit. Must be an array with shape (qubits, layers, dimension of data)
268     -data: one data for training. It must be (x,y)
269     -reprs: variable encoding the label states of the different classes
270     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
271     OUTPUT:
272     -chi^2 for data
273     """
274
275     #
276     x, y = data
277     print('data=', data)
278     theta_aux = code_coords(theta, alpha, x)
279     C = circuit(theta_aux, entanglement)
280
281     '''if y==0:
282         s=np.array([0,0,-1])
283     elif y==1:
284         s=np.array([0,0,1])
285     elif y==2:
286         s=np.array([1,0,0])
287     elif y==3:
288         s=np.array([-1,0,0])
289     elif y==4:
290         s=np.array([0,1,0])
291     elif y==5:
292         s=np.array([0,-1,0])'''
293     ans = trace_dis(reprs[y], C.r)
294     return ans
295
296
297 #Sara
298 #####
299 def Av_Tr(theta, alpha, train_data, reprs, entanglement): #Chi in average
300     """
301     This function compute chi^2 for only one point
302     INPUT:
303     -theta: set of parameters needed for the circuit. Must be an array with shape (qubits, layers, 3)
304     -alpha: set of parameters needed for the circuit. Must be an array with shape (qubits, layers, dimension of data)
305     -data: one data for training. It must be (x,y)
306     -reprs: variable encoding the label states of the different classes
307     -entanglement: whether there is entanglement or not in the Ansätze, just 'y'/'n'
308     OUTPUT:
309     -Averaged chi^2 for data
310     """
311     Av_Tr = 0
312     for d in train_data:
313         Av_Tr += _Tr(theta, alpha, d, reprs, entanglement)
314
315     return Av_Tr / len(train_data)
316
317
318 #####
319
320
321
322
323
324
325
326
327
328
```


References

- 1 Pérez-Salinas, A., Cervera-Lierta, A., Gil-Fuster, E. & Latorre, J. I. Data re-uploading for a universal quantum classifier. *Quantum* **4**, 226 (2020).
- 2 Schuld, M., Sinayskiy, I. & Petruccione, F. An introduction to quantum machine learning. *Contemporary Physics* **56**, 172-185 (2015).
- 3 Schuld, M., Bocharov, A., Svore, K. M. & Wiebe, N. Circuit-centric quantum classifiers. *Physical Review A* **101**, 032308 (2020).
- 4 LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *nature* **521**, 436-444 (2015).
- 5 Li, W. & Deng, D.-L. Recent advances for quantum classifiers. *Science China Physics, Mechanics & Astronomy* **65**, 220301 (2022).
- 6 Helstrom, C. W. Quantum detection and estimation theory. *Journal of Statistical Physics* **1**, 231-252 (1969).
- 7 Nielsen, M. A. & Chuang, I. L. *Quantum computation and quantum information*. (Cambridge university press, 2010).
- 8 Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks* **4**, 251-257 (1991).
- 9 Cerezo, M., Verdon, G., Huang, H.-Y., Cincio, L. & Coles, P. J. Challenges and opportunities in quantum machine learning. *Nature Computational Science* **2**, 567-576 (2022).
- 10 Liu, D. C. & Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical programming* **45**, 503-528 (1989).
- 11 Zhu, C., Byrd, R. H., Lu, P. & Nocedal, J. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)* **23**, 550-560 (1997).
- 12 Liu, Y. *et al.* On centroidal Voronoi tessellation—energy smoothness and fast computation. *ACM Transactions on Graphics (TOG)* **28**, 1-17 (2009).
- 13 Wang, L., Zhou, K., Yu, Y. & Guo, B. Vector solid textures. *ACM Transactions on Graphics (TOG)* **29**, 1-8 (2010).
- 14 Byrd, R. H., Lu, P., Nocedal, J. & Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing* **16**, 1190-1208 (1995).
- 15 Virtanen, P. *et al.* SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* **17**, 261-272 (2020).
- 16 Bonet-Monroig, X. *et al.* Performance comparison of optimization methods on variational quantum algorithms. *Physical Review A* **107**, 032407 (2023).
- 17 Pellow-Jarman, A., Sinayskiy, I., Pillay, A. & Petruccione, F. A comparison of various classical optimizers for a variational quantum linear solver. *Quantum Information Processing* **20**, 202 (2021).
- 18 Nelder, J. A. & Mead, R. A simplex method for function minimization. *The computer journal* **7**, 308-313 (1965).
- 19 Gao, F. & Han, L. Implementing the Nelder-Mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications* **51**, 259-277 (2012).
- 20 Abel, S., Blance, A. & Spannowsky, M. Quantum optimization of complex systems with a quantum annealer. *Physical Review A* **106**, 042607 (2022).
- 21 Lockwood, O. An empirical review of optimization techniques for quantum variational circuits. *arXiv preprint arXiv:2202.01389* (2022).
- 22 Kraft, D. A software package for sequential quadratic programming. *Forschungsbericht- Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt* (1988).